UNIVERSITY OF KHARTOUM

FACULTY OF ENGENEERING AND ARCHITHCTURE

ELECTERICAL AND ELECTRONIC ENGENEERING DEPARTMENT

COMMUNICATION IN DISTRIBUTED SYSTEMS

A thesis submitted for partial Fulfillment of the requirements of the

Degree of Master in Telecommunication and Information Systems

By: Rusha Mohamed Hilo Omer.

Supervisor:Dr.Elrashid Osman Khider.

2006

ACKNOWLEDGMENT

I am deeply indebted to my supervisor Dr. Alrasheed Osman Khider, Faculty of Engineering and Architecture for his patience, guidance and valuable criticism and encouragement.

I am also indebted to Osama Abdulla Obeid , Abdelgadir Mahmoud, and Haram Ahmaed Mohamed for the valuable help they offered during the preparation of the manuscript.

My thank go to my father for continuous encouragement.

Finally, I am indebted to University of Science and Technology for offering me the chance to prepare for this degree.

To my Father

&Mother

who

Really supported me

LIST OF CONTENTS

Abstract …… … i

المستخلص ....…… …… II

Chapter 1:Introduction

1.1 Background of Distributed Systems……....1

1.2.Problem Statement …… ...2

1.3.Objective…….....2

1.4.Thesis layout…….....2

Chapter 2: Literature Review of Communication in Distributed Systems

2.1.Computer systems……....4

2.2.A distributed computing system…… ……..4

2.3.Goal of a distributed operating system……...…….....4

2.4. Applications for Distributed Computer Systems…….5

2.5.A Disadvantages of Distributed Systems…….5

2.6.Problems in the design of a distributed operating system…….....5

Chapter 3:Materials and Methods

3.1. Hardware Requirement……..….. ...6

3.2. Software Requirements…….. 7

3.3.Methods……...…. 7

3.3.1.System Calls()……...…..... 7

3.3.2.User Defined Function……..…...…....7

3.3.3.System component……...……7

3.4. Methods for Client-Server Model……..…….7

3.5. Methods for Remote Procedure Call programming……..…..9

3.6. Methods for Group Communication Program……..……..…….. 9

Chapter 4: Network Communication Protocols

4.1.The OSI Reference Model……..…… . .11

4.1.1. Layered Architecture…….…….....12

4.1.2.Relationships Among OSI Reference Model Layers…….……....12

4.1.3.Layer Definition……..…… 14

4.1.3.1.Application Layer…….……...14

4.1.3.2.Presentation Layer…… ……..14

4.1.3.3.Session Layer……..……....14
4.1.3.4.Transport Layer…….. ……..15
4.1.3.5.Network Layer……..……..…..15
4.1.3.6.Data-Link Layer……15
4.1.3.7.Physical Layer…… … ….... .16
4.2.TCP/IP Protocol……...……..16
4.3.TCP/IP and OSI……..17
4.3.1.Layer Definition……17
4.3.1.1.Network Interface Layer…….....17
4.3.1.2. Internet Layer…….…....17
4.3.1.3.Transport Layer…….……..17
4.3.1.4.ApplicationLayer……..… …..18
Chapter 5:The Client-Server Model
5.1. Clients and Servers…….....19
5.2.Issues in Client-Server Communication……...20
5.2.1. Addressing…… 20
5.2.2. Blocking versus Nonblocking Primitives……. 22
5.2.3. Buffered versus Unbuffered Primitives…….…… 23
5.2.4. Reliable versus Unreliable Primitives……24
5.2.5. Implementing the Client-Server Model…….…… 25
Chapter 6:Remote Procedure Call(RPC)
6.1.How RPC works……..... 27
6.2.Marshaling and Data types…… 28
6.3.Finding the Server…… 29
6.4.RPC Semantics in the Presence of Failures…… 31
6.4.1.Client Cannot Locate the Server…….. 31
6.4.2.Lost Request Messages…… ...31
6.4.3.Lost Reply Messages……...31
6.4.4.Server Crashes…….32
6.4.5.Client Crashes…….....32
6.5.Implementation Issues for RPC System, with a special emphasis on the performance…… 33
6.5.1.Acknowledgements…… 33
6.5.2.Critical path…… 33
6.5.3.Timer Management…… 34

6.5.4.Copying…… 35

Chapter 7:Group Communication

7.1.Group Communications-the Basic…… …… 36

7.2.Design Issues for Group Communications……...…… 36

7.2.1.Closed Groups versus Open Groups…….. …… 37

7.2.2.Peer Group versus Hierarchical Group……...… 37

7.2.3.GroupMembership…… . 38

7.2.4.Grou Addressing……..… 38

7.2.5.Atomicity… …… 39

7.2.6.MessageOrdering…….. ...40

7.2.7.OverlappingGroups……. …41

Chapter 8: Design for Communication in Distributed Systems..

8.1. Client-Server model programming ……..…… … 42

8.2.Socket Types……..…… 43

8.3.The C code for client server communication…….…… .43

8.4.Remote Procedure Call Programming…….…….. 46

8.5.Group Communication Programming ……. ……. 48

8.6. Communicating with Multicast Group……..51

8.7. Results..... ….51

8.7.1.Client-Server model programming……...…… … ..51

8.7.2.Remote Procedure Call programming……..…… …..52

8.7.3.Group Communication programming……. ……..…… ……...53

Chapter 9: Analysis of the program

9.1. Client-Server model……...…… …… …. …..55

9.2. Remote Procedure Call……...…… … …..…… … . …..58

9.3. Group Communication……...…… …… .……60

Chapter 10:Conclusions and Recommendation

10.1.Conclusions…… …… …… 61

10.2.Recommendation……..…….…… 61

Appendices

Appendix(A)…… 63

Appendix(B)…… …… 67

Appendix(C)…….…… 71

References……..…… ..…….…… 74

LIST OF FIGURES

Fig. 3.1. Communication elements…… 6
Fig.4-1. The seven-layer OSI reference model…… 12
Fig.4-2. Relationships among OSI layers…….. 13
Fig.5-1. A typical message as it appears on the network…… 20
Fig.5-2.(a)Machine.process addressing.(b)Process addressing with broadcasting.(c)Address lookup via server ……. 20
Fig.5-3. (a)A blocking send primitive. (b)A nonblocking send primitive..... 23
Fig.5-4.(a) Unbuffered message passing . (b) buffered message passing…… 24
Fig.5-5. (a) Individually acknowledged messages. (b) Reply being used as the acknowledgement of the request. Note the ACKs are handled entirely within the kernels…… 25
Fig.5-6. Four design issues for the communication primitives and some of the principal choices available…… 26
Fig.6-1. Calls and messages in an RPC. Each ellipse represents a single process, with the portion being the stub…… 28
Fig.6-2. The binder interface……. 30
Fig.6-3.(a) Normal case. (b) Crash after execution. (c) Crash before execution…… 32
Fig.6-4. Critical path from client to server…… 34
Fig.7-1.(a) Outsiders may not send to a closed group.(b) Outsiders may send to an open group…. 37
Fig.7-2. (a) Communication in a peer group. (b) Communication in a simple hierarchical group… 37
Fig.7-3. Process 0 sending to a group consisting of processes 1,3, and 4, (a) Multicast implementation.(b) Broadcast implementation. (c) Unicasts implementation…….. 39
Fig.7-4.(a) The three messages sent by processes 0 and 4 are interleaved in time. (b) Graphical representation of the six messages, showing the arrival order…….. 40
Fig.7-5. Four processes, A, B, C, and D, and four messages. Processes B and C get the messages from A and D in a different order…… 41
Fig.8-1. Flow chart of the client code in client-server model programming.. ……...….44
Fig.8-2. Flow chart of the server code in client-server model programming.. ……...…46

Fig.8-3. Flow chart of the client code in remote procedur call programing……..47

Fig.8-4. Flow chart of the server code in remote procedur call programing…… 48

Fig.8-5. flow chart of the Multicastsniffer code in group communication programming…..49

Fig.8-6. flow chart of the Multicastsender code in group communication programming .. 51

Fig.8-7. Client result…… 51

Fig.6-8. Server result……. . 52

Fig.8-9. Add result(Client result)…… 52

Fig.8-10. Subtract result(Client result)…… 53

Fig.8-11. Server result…… 53

Fig. 8-12. Multicastsniffe Result…….. 54

Fig. 8-13. Multicastsender Result…… 54

Abstract

The past 1980's witnessed two advances in technology. The first was the development of powerful microprocessors. The second invention was the high speed local area networks or LANs. These two developments permitted putting together computing systems composed of large numbers of CPUs connected by a high-speed network, thus allowing transfer of information between machines in milliseconds. Such system is called distributed system.

This thesis introduces three mechanisms of communication in distributed systems. The client server model is designed to support the massage exchange. The remote procedure call allows client programs to call procedures in server programs running in separate processes and generally in a different computer from the client. The group communication can handle communication mechanisms in which a massage can be sent to multiple receivers in one operation.

The work aim at designing two programs for communication using C-language under Linx:

- Client Server model
- Remote Procedure Call

For group communication program java- language has been a adopted.

المستخلص

شهدت الثمانينات من القرن الماضى تقدما تكنولوجيا فى مجالين. كان الاول استحداث المعالج الدقيق. اما التطور الثانى الشبكات المحليه ذات السرعه العاليه. هذان التطوران سمحا بوضع انظمه تحليليه متكامله مكونه من عدد كبير من وحدات المعالجه المركزيه مرتبطه بواسطه شبكه ذات سرعه عاليه, مما يسمح بنقل المعلومات بين الاجهزه بكفاءه عاليه. هذا النظام يدعى نظام موزع.

هذا المشروع يقدم ثلاثه آليات اتصال عن الانظمه الموزعه. نموذج العميل/ المخدم مصمم لدعم تبادل الرساله بين العميل و المخدم. اجراء الاستدعاء عن بعد يسمح لبرامج العميل باستدعاء اجراءات فى برامج المخدم تنفذ فى معالجات متفرقه وعموما فى حواسيب مختلفه من العميل. الاتصال الجماعى يمكن ان يتعامل بآليه اتصال بحيث يمكن ان ترسل رساله واحده لعدد من المستقبلين بعمليه واحده.

يهدف هذا البحث لتصميم برنامجين للاتصال باستخدام لغه C تحت ال LINX للاتى :

- نموذج العميل/ المخدم.
- اجراء الاستدعاء عن بعد.

اما بالنسبه للاتصال الجماعي فقد تم تصميمه بلغه ال java.

Chapter One

Introduction

Chapter 1

Introduction

1.1 Background of Distributed Systems

The use of computers is in the process of undergoing a revolution. From 1945, when the compute began, until about 1985, computers were large and expensive. Even minicomputers normally cost tens of thousands of dollars each. As a result, most organizations had only a handful of computers, and for lack of a way to connect them, these usually operated independently from one another.

Starting in the mid 1980, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors.

The second development was the invention of high-speed local area networks or LANs. These systems allowed dozens, or even hundreds, of machines to be connected in such a way that small amounts of information can be transferred between machines in a millisecond or so. Larger a mounts of data can be moved between machines at rates of 10 million bit /sec and more.

The net result of these two technologies is that it is now only feasible, but easy to put tog there computing systems composed of large n numbers of CPUs connected by high-speed network. They are usually called distributed systems, in contrast to the previous centralized systems consisting of a single CPU, its memory, peripherals, and some terminals.

A distributed system is a collection of processors that don't share memory or a clock, instead, each processor communicate with one another through various communication networks, such as high speed buses or telephone lines, the processors in a distributed system may very in size and function, they may include small microprocessors, workstations, minicomputers and large general purpose computer systems, these processors are referred to by a number of names.

Distributed systems are attractive mainly due to the increased ability for resource sharing, computation speed-up, reliability, communication and economics. In distributed system there is no shared memory.

Due to the a absence of shared memory all communication between processes is achieved by means of messages. Network communication depends on two autonomous processes existing at the same time and on their agreeing to communicate. Cooperative communication can succeed only if the two processes agree on the precise syntax and semantic of the information to exchanged. To

exchange information between computers there are many different network protocols to address the range of communications applications. The term protocol is used to a well-known set of rules and format to be used for communication between processes in order to perform a given task.

1.2.Problem Statement

With so many factors to synchronize, a great deal of coordination across the nodes of a network is necessary if communication is to occur at all, let alone accurately or efficiently. A single manufacturer can build all of its products to work well together, but if some of the best components for your needs are not made by the same company difficulties arise.

Standards are essential in creating and maintaining an open and competitive market for equipment manufacturers and in guaranteeing national and international interoperability of data and telecommunications technology and processes. They provide guidelines to manufacturers, vendors, government agencies, and other service providers to ensure the kind of interconnectivity necessary in today's marketplace and international communications.[3]

Message passing very flexible but

- Requires that programmer worries a bout message format.
- Messages must be packed and unpacked.
- Messages have to be decoded by servers to figured what is requested.
- Messages are often asynchronous.
- They may require special error handling functions.

1.3.Objective

The main objective of this thesis is concerned with software for communication in distributed systems. In this thesis I shall study:

1. Client-Server model: discuss design in message passing which concern the communication primitives.
2. Remote procedure call(RPC): discuss the works of RPC.
3. Group communication: that will discuss a design for group communication, show that it can be implemented efficiently.

The thesis describes the implementation of application of the three mechanism of communication.

1.4.Thesis layout

Chapter 2 introduces a brief outline of computer systems, distributed computing systems, the goal of distributed operating systems, applications for distributed computer systems, disadvantages of distributed systems and problems in the design of a distributed operating system.

Chapter 3 details materials and methods used in the thesis.

Chapter 4 introduces the network communication protocol. A protocol is implemented by a pair of software modules located in the sending and receiving computers. A need arose for standard protocols that could allow hardware and software from different vendors to communicate. In response, two primary sets of standards were developed: the OSI reference model and TCP/IP are the most common protocols used in system communication. The problem with all these layers, is that all the layers add overhead. This problem especially noticeable in system with high speed interconnects where the processing time counts.

Chapter 5 of the thesis deals with client and server. This model allows large application to be split into smaller tasks and perform the tasks among server machine and client machine in the network..

Chapter 6 explains in steps how to perform a remote procedure call (RPC). The problems and failures faced and how to deal with them is mentioned.

Chapter 7 deals with group communication, which is in essence sending a single message to multiple processes. Group can be organized in multiple ways and can be addressed in many ways. The chapter deals with different facets and organizational aspects of group communication.

Chapter 8 design for communication in distributed systems communication between client and server, remote procedure call programming, group communication programming and results of the programs.

Chapter 9 analysis of the program

Chapter 10 introduces conclusions and recommendations.

Chapter Two

Literature Review of Communication in Distributed Systems

Chapter 2

Literature Review of Communication in Distributed Systems

2.1.Computer systems

Computer systems consisting of multiple processors are becoming commonplace. Many companies and institutions, for example, own a collection of workstations connected by a local area network (LAN). Although the hardware for distributed computer systems is advanced, the software has many problems. We believe that one of the main problems is the communication paradigms that are used. Distributed computer systems, or distributed systems for short, consist of hardware and software components. The term ‘‘distributed system’’ is used for many hardware configurations.

2.2.A distributed computing system

A distributed computing system consists of multiple autonomous machines
that do not share primary memory, but cooperate by sending messages over
a communication network. This definition includes architectures like a set of workstations connected by a LAN, or a set of computers geographically distributed over the world connected by a wide-area network (WAN). The key property of a distributed system is that there is no physically shared memory present.Software for distributed systems can be divided between application software and system software.

2.3.Goal of distributed operating system

The goal of a distributed operating system is to make a collection of computers
look like a single computer to the user. It is the operating system’s responsibility to allocate resources to users in a transparent way]. To understand what distributed operating systems are, it is useful to compare them with the more common *network operating systems*. In a network system each user has a computer for his exclusive use. The computer has its own operating system and may have its own disks. All commands are normally run locally on the user’s computer. If a user wants to copy a remote file, he explicitly has to tell where the file is located. If a user wants to start a program on another computer, he has to start the command explicitly on the remote computer. More advanced network systems may provide some degree of transparency by, for example, having a shared file system, but the key point is that the user is always aware of the distribution of the hardware. Unlike true distributed systems, network systems are very common. Both network and distributed systems are an active area of research. One of the main

problems is that existing network and distributed operating systems do not provide adequate support for writing distributed applications.

2.4. Applications for Distributed Computer Systems

Applications that make use of a distributed system can be categorized in four different types: functional specialization, parallel applications, fault-tolerant applications, and geographically distributed applications .

Some distributed applications can be written as a set of specialized services. For example, a distributed operating system may be written as a collection of services, such as a file service, a directory service, a printer service, and a time-of-day service. In a distributed system it makes sense to dedicate one or more processors to each service to achieve high performance and reliability. Because distributed systems consist of multiple processors, they can also be used to run parallel applications. The goal in a parallel application is to reduce the turnaround time of a *single* program by splitting the problem into multiple tasks, all running in parallel on separate processors. Distributed systems are potentially more reliable than centralized systems, because each processor is independent; if one fails, it need not affect the others. Reliability can therefore be increased by replicating an application on multiple processors. If one of the processors fails, another one can finish the job. Finally, there are applications which are inherently distributed in nature.

2.5.A Disadvantages of Distributed Systems

The disadvantages, chief among them is the complexity and relative unavailability of software. Most of the problems of non-distributed systems still exist, but we have added many new problems to solve in order to achieve some of the perceived advantages among them are network congestion and rerouting and security. Also most distributed systems complicate the job of the user by forcing them to be aware of various aspects of the distributed system .[3]

2.6.Problems in the design of a distributed operating system

- Communication Primitive
- Naming and Protection
- Resource Management
- Fault Tolerance
- Services to provide

Chapter 3

Materials and Methods

Chapter 3

Materials and Methods

The requirements of case study can be divided into two parts:

3.1. Hardware Requirement:

To build a distributed system, some of hardware requirement had to be provided that makes the system able to work. Fig2.1.represents communication elements. These components can be explained as follow:

- **Clients:** The clients are computers that request a service or access information stored on devices such as servers. Many clients can be connected to other devices on the network such as servers using any type of protocol through an Internet network.
- **Servers:** The servers are computers that provide the required information and resources available to other computers on a network. Each server can be connected to many clients or to many other servers. The whole connection carried out using any type of networks.
- **Media:** The wires that make the physical connections.
- **Shared data:** Files provided to clients by servers across the network.
- **Shared printers and other peripherals:**Additional resources provided by servers.
- **Resources:**Any service or device, such as files, printers, or other items, made available for use by members of the network.

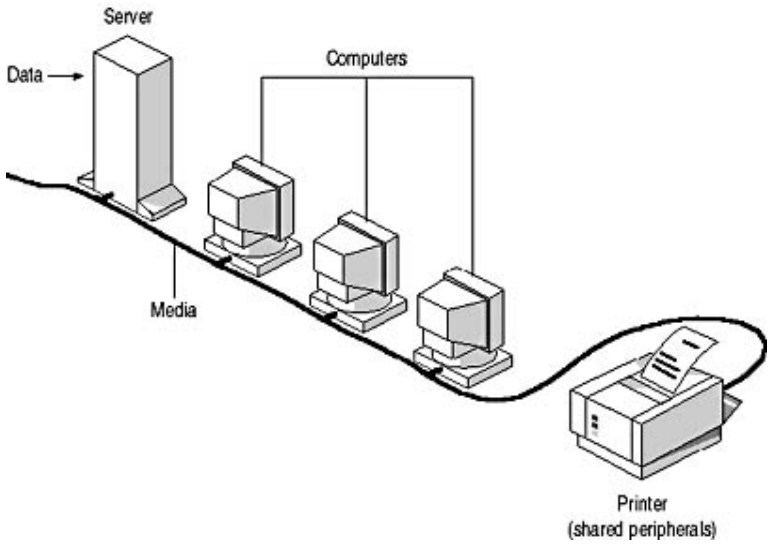


Figure 3.1. Communication elements

3.2. Software Requirements:

To represent the system, many software requirements had to be provided like programs. In this case the needed software programs that must be provided are; LINX and Windows (XP, Millennium, Win98, etc), Internet Explorer and programming language. There are of course, many languages which can be used, **JAVA** and **C** language were chosen to program the present communication in distributed system design. Used sockets for three programs. Both client-server model and RPC programming C language was used ware as for group communication JAVA language was used.[4]

3.3.Methods

3.3.1.System Calls()

System calls are functions that a programmer can call to perform the services of the operating system. Some of them involve access to data that users must not be permitted to corrupt or even change and they are the only way to access kernel facilities such as the file system, the multitasking mechanisms, and the interprocess communication primitives.

3.3.2.User Defined Function

A user defined function defined by the programmer and used when the user need to run a group of command or to perform a specific task, that can't be done by one command or system call.[8]

3.3.3.System component

- Client-server model software is composed of two programs, server program, and client program.
- Remote procedure call is composed of three programs, the interface file, program, and client program.
- Group communication composed of two programs, Multicast Sender program, and Multicast Sniffer program.

3.4. Methods for Client-Server Model
Server Program

The following system calls that I have used in my server code to establishing a socket:

- Perror()

  This routine is called when a system call fails. It displays a message(it receives as an argument) about the error on stderr and then aborts the program.

- Socket()

  The Socket() system call creates a socket.

- atoi()

Function to convert this from sting of digits to an integer

- htons()

  Function to converts a port number in host byte order to a port number in network byte order.

- Bind()

  system call binds a socket to an address.

- Listen()

  Is system call allows the process to listen on the socket for connections.

- Accept()

  Is system call causes the process to block until a client connect to the server.

- Fork()

  Fork() system call is used to create a new process.

- Dostuff()

  Is a dummy function this function will handle the connection after it has been established and provide whatever services the client requests.

- Read()

  Is system call is used to copy an arbitrary number of characters or byte from a file into a buffer under the control of the calling program.

- Write()

  Is system call is natural inverse of read. It copies data from a program buffer, to an external file.

- Close()

  The normal unix close function is also used to close a socket and terminate TCP connection.

Client Program

In addition to the server system calls read(),write(),atoi(),perror(),socket() in our server program there are new system calls:

- Gethostbyname()

  Takes the host's name and finds its corresponding dotted string IP address.

- Connect()

  The connect() function is called by the client to establish a connection to server.

3.5. Methods for Remote Procedure Call programming

Server Program

- svctcp_create()

  Used for a server creation rotine.

- svc_register()

  Used for a server registration rotine.

- svc_run()

  Is used to disbatcher for the RPC library and to call the remote procedures in response to RPCrequests and decodes remote procedure arguments and encodes results.

- svc_sendreply()

  For sending a reply to RPC call

- svcerr_noproc()

  Is used when finding error in the procedur requested by the RPC.

- svcerr_decode()

  Used when finding error in decoding remote procedure arguments.

- svc_freeargs()

  After using the character array, you can free it with svc_freeargs, when called from svc_freeargs the memory deal locator is used.

Client Program

- clnt_create()

  create client"handel" used for calling MCPROG on the server designated on the command line. Use the TCP protocol when contacting the server.

- clnt_pcreateerror()

  an error occurred while calling the server. Print error message and stop. Couldn't establish connection with server.

- clnt_destroy()

  The connection between client and server will remain in place until clnt_destroy() is called.

3.6. Methods for Group Communication Program

The following system calls that I have used in my group communication program

- Multicastsender()

  Class that sends data to the address that read from the command line to a multicast group.

- GetBytes()

  tostuffs the string(some data)into the byte array data.

- MulticastSocket( )

To creates a MulticastSocket ms on the specified port.

- ms.joinGroup( )

  To join the multicast group at specified intern address.

- Send( )

  MulticastSocket sends the datagram packet to the group by send system call.

- Receive( )

  To receive the data by calling receive.

- Leave Group( )

  When you no longer want to receive data, you leave the multicast group by invoking the socket's Leave Group( ) method.

Chapter 4

Network Communication Protocols

Chapter 4

Network Communication Protocols

Network activity involves sending data from one computer to another. This complex process can be broken into discrete, sequential tasks. The sending computer must:

1. Recognize the data.
2. Divide the data into manageable chunks.
3. Add information to each chunk of data to determine the location of the data and to identify the receiver.
4. Add timing and error-checking information.
5. Put the data on the network and send it on its way.

Network client software operates at many different levels within the sending and receiving computers. Each of these levels, or tasks, is governed by one or more protocols. These protocols, or rules of behavior, are standard specifications for formatting and moving the data. When the sending and receiving computers follow the same protocols, communication is assured. Because of this layered structure, this is often referred to as the protocol stack.

4.1.The OSI Reference Model

In 1984, the (Open Systems Interconnection) OSI was created by the International Organization for Standardization (ISO) .The OSI has become an international standard and serves as a guide for networking.

The OSI reference model is the best-known and most widely used guide for visualizing networking environments. Manufacturers adhere to the OSI reference model when they design network products. It provides a description of how network hardware and software work together in a layered fashion to make communications possible. The model also helps to troubleshoot problems by providing a frame of reference that describes how components are supposed to function.

4.1.1. Layered Architecture

The OSI reference model architecture divides network communication into seven layers. Each layer covers different network activities, equipment, or protocols. Fig.4-1. represents the layered architecture of the OSI reference model. (Layering specifies different functions and services as data moves from one computer through the network cabling to another computer.) The OSI reference model defines how each layer communicates and works with the layers immediately above and below it. For example, the session layer communicates and works with the presentation and transport layers.

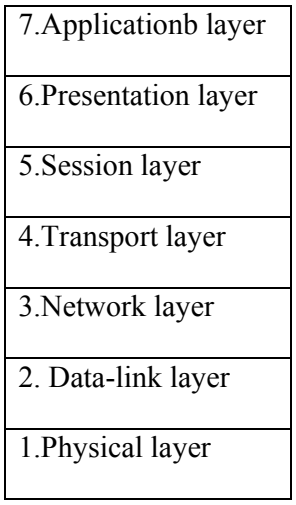


Fig.4-1. The seven-layer OSI reference model

Each layer provides some service or action that prepares the data for delivery over the network to another computer. The lowest layers-1 and 2-define the network's physical media and related tasks, such as putting data bits onto the network interface cards and cable. The highest layers define how applications access communication services. The higher the layer, the more complex its task.

The layers are separated from each other by boundaries called interfaces. All requests are passed from one layer, through the interface, to the next layer. Each layer builds upon the standards and activities of the layer below it.

4.1.2.Relationships Among OSI Reference Model Layers

Each layer provides services to the next-higher layer and shields the upper layer from the details of how the services below it are actually implemented. At the same time,

each layer appears to be in direct communication with its associated layer on the other computer. This provides a logical, or virtual, communication between peer layers, as shown in Fig.4-2. In reality, actual communication between adjacent layers takes place on one computer only. At each layer, software implements network functions according to a set of protocols.

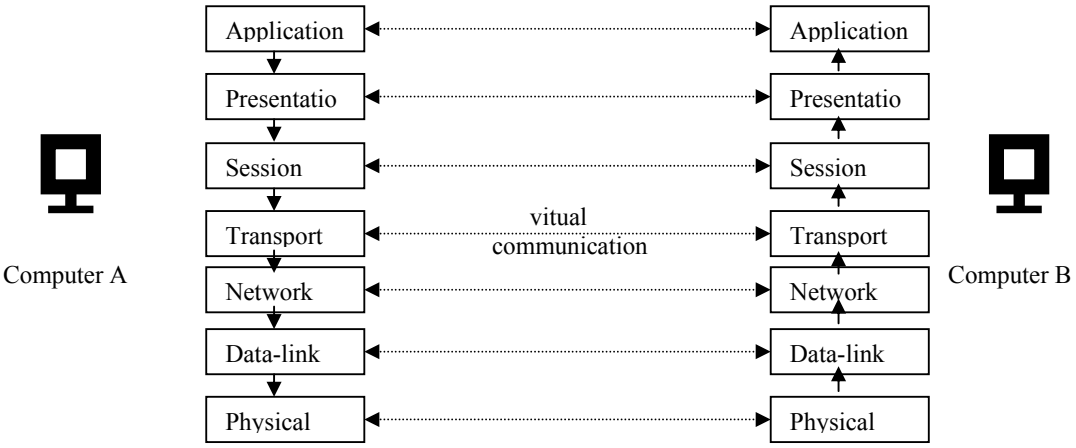


Fig.4-2. Relationships among OSI layers

Before data is passed from one layer to another, it is broken down into packets, or units of information, which are transmitted as a whole from one device to another on a network. The network passes a packet from one software layer to another in the same order as that of the layers. At each layer, the software adds additional formatting or addressing to the packet, which is needed for the packet to be successfully transmitted across the network.

At the receiving end, the packet passes through the layers in reverse order. A software utility at each layer reads the information on the packet, strips it away, and passes the packet up to the next layer. When the packet is finally passed up to the application layer, the addressing information has been stripped away and the packet is in its original form, which is readable by the receiver.

. With the exception of the lowest layer in the OSI networking model, no layer can pass information directly to its counterpart on another computer. Instead, information on the sending computer must be passed down through each successive layer until it reaches the physical layer. The information then moves across the networking cable to the receiving computer and up that computer's networking layers until it arrives at the corresponding layer. For example, when the network layer sends information from computer A, the information moves down through the data-link and physical layers on the sending side, over the cable, and up the physical and data-link layers on the receiving side to its final destination at the network layer on computer B.

In a client/server environment, an example of the kind of information sent from the network layer on computer A to the network layer on computer B would be a network address, with perhaps some error-checking information added to the packet.

Interaction between adjacent layers occurs through an interface. The interface defines the services offered by the lower networking layer to the upper one and further defines how those services will be accessed. In addition, each layer on one computer appears to be communicating directly with the same layer on another computer.[1]

4.1.3. Layer Definition

4.1.3.1.Application Layer

Layer 7, the topmost layer of the OSI reference model, is the application layer. This layer provides access to the OSI environment for users and also provides distributed information services. This layer contains management function and generally useful mechanisms to support distributed applications. In addition, general-purpose applications such as file transfer, electronic mail, and terminal access to remote computers are considered to reside at this layer.

4.1.3.2.Presentation Layer

Layer 6, the presentation layer, defines the meaning of the bits in a message. Until this layer, any messages (or packets at lower layers), are just a pile of bits, no mechanism is provided below layer 6 to make sense of the contents. Of course, when you are only concerned with getting the correct bits down the wire, it doesn't matter what the bits mean.

4.1.3.3.Session Layer

Layer 5, the session layer, allows two applications on different computers to open, use, and close a connection called a session. It is involved with synchronization and checkpointing. This layer is frequently omitted. Its job is to allow recovery of transmission if an interruption causes some of the packets to be lost. Without this layer, the entire message must be retransmitted, even if all but one packet were received correctly.

4.1.3.4.Transport Layer

Layer 4, the transport layer, provides an additional connection level beneath the session layer. The transport layer ensures that packets are delivered error free, in sequence, and without losses or duplications. At the sending computer, this layer repackages messages, dividing long messages into several packets and collecting small packets together in one package. This process ensures that packets are transmitted efficiently over the network. At

the receiving computer, the transport layer opens the packets, reassembles the original messages, and, typically, sends an acknowledgment that the message was received. If a duplicate packet arrives, this layer will recognize the duplicate and discard it.

The transport layer provides flow control and error handling, and participates in solving problems concerned with the transmission and reception of packets. Transmission Control Protocol (TCP) and Sequenced Packet Exchange (SPX) are examples of transport-layer protocols.

4.1.3.5.Network Layer

Layer 3, the network layer, is responsible for addressing messages and translating logical addresses and names into physical addresses. This layer also determines the route from the source to the destination computer. It determines which path the data should take based on network conditions, priority of service, and other factors. It also manages traffic problems on the network, such as switching and routing of packets and controlling the congestion of data.

If the network adapter on the router cannot transmit a data chunk as large as the source computer sends, the network layer on the router compensates by breaking the data into smaller units. At the destination end, the network layer reassembles the data. Internet Protocol (IP) and Internetwork Packet Exchange (IPX) are examples of network-layer protocols.

4.1.3.6.Data-link Layer

This layer validates that the correct bits have been received. This is typically accomplished through the use of a checksum. The checksum is included in the layer 2 header for a packet and is validated by the layer 2 protocol at the receiving end. If the checksum does not match, a request is sent back asking for a resend of the packet. The hardware device used for Layer 2 is called a bridge. A bridge cannot handle multiple paths. It only knows if an outgoing packet is local or not local. If not local, the bridge allows the packet to pass through. For incoming packets, if the packet is not local, the bridge throws it away, if local, the packet passes through.

4.1.3.7.Physical Layer

Layer 1, the bottom layer of the OSI reference model, is the physical layer. This layer transmits the unstructured, raw bit stream over a physical medium (such as the network cable). The physical layer is totally hardware-oriented and deals with all aspects of

establishing and maintaining a physical link between communicating computers. The physical layer also carries the signals that transmit data generated by each of the higher layers. It defines how many pins the connector has and the function of each. It also defines which transmission technique will be used to send data over the network cable. This layer provides data encoding and bit synchronization. The physical layer is responsible for transmitting bits (zeros and ones) from one computer to another, ensuring that when a transmitting host sends a 1 bit, it is received as a 1 bit, not a 0 bit. Because different types of media physically transmit bits (light or electrical signals) differently, the physical layer also defines the duration of each impulse and how each bit is translated into the appropriate electrical or optical impulse for the network cable.

4.2.TCP/IP Protocol

Transmission Control Protocol/Internet Protocol (TCP/IP) has become the standard protocol used for interoperability among many different types of computers. This interoperability is a primary advantage of TCP/IP. Most networks support TCP/IP as a protocol. TCP/IP also supports routing and is commonly used as an internetworking protocol.

Other protocols written specifically for the TCP/IP suite include:

- SMTP (Simple Mail Transfer Protocol) E-mail.
- FTP (File Transfer Protocol) For exchanging files among computers running TCP/IP.
- SNMP (Simple Network Management Protocol) For network management.

Designed to be routable, robust, and functionally efficient, TCP/IP was developed by the United States Department of Defense as a set of wide area network (WAN) protocols. Its purpose was to maintain communication links between sites in the event of nuclear war. The responsibility for TCP/IP development now resides with the Internet community as a whole.

4.3.TCP/IP and OSI

The TCP/IP protocol does not exactly match the OSI reference model. Instead of seven layers, it uses only four. Commonly referred to as the Internet Protocol Suite, TCP/IP is broken into the following four layers:

- Network interface layer

- Internet layer
- Transport layer
- Application layer

Each of these layers corresponds to one or more layers of the OSI reference model.

4.3.1. Layer Definition

4.3.1.1.Network Interface Layer

The network interface layer, corresponding to the physical and data-link layers of the OSI reference model, communicates directly with the network. It provides the interface between the network architecture (such as token ring, Ethernet) and the Internet layer.

4.3.1.2.Internet Layer

The Internet layer, corresponding to the network layer of the OSI reference model, uses several protocols for routing and delivering packets. They function at this layer of the model and are used to forward packets from one network or segment to another. Several protocols work within the Internet layer.

4.3.1.3.Transport Layer

The transport layer, corresponding to the transport layer of the OSI reference model, is responsible for establishing and maintaining end-to-end communication between two hosts. The transport layer provides acknowledgment of receipt, flow control, and sequencing of packets. It also handles retransmissions of packets. The transport layer can use either TCP or User Datagram Protocol (UDP) protocols depending on the requirements of the transmission.

4.3.1.4.Application Layer

Corresponding to the session, presentation, and application layers of the OSI reference model. This is basically concerned with defining the protocols. In practice it works by sending an unbroken " data stream " to the transport layer. At this stage the stream is broken into packets, each of which is "framed " with a TCP header (which contains the sender's and recipient's addresses and error checking information).[4]

Chapter 5

The Client-Server Model

Chapter 5

The Client-Server Model

5.1.Clients and Servers

The main emphasis of Client-Server architecture is to allow large application to be split into smaller tasks and to perform the tasks among host (server machine) and desktops (client machine) in the network.

The goals of Client-Server Computing are to allow every networked workstation (Client) and host (Server) to be accessible, as needed by an application, and to allow all existing software and hardware components from various vendors to work together. When these two conditions are met, the environment can be successful and the benefits of client/server computing, such as cost savings, increased productivity , flexibility, and resource utilization, can be realized.

In this model the operating system structure as a group of cooperating processes, called servers , that offer services to the users ,called clients .

A Client Machine usually manage the user-interface portion of the application, validate data entered by the user, dispatch requests to server programs. It is the front-end of the application that the user sees and interacts with. Besides, the Client Process also manages the local resources that the user interacts with such as the monitor, keyboard, workstation, CPU and other peripherals.

The Server Machine fulfills the client request by performing the service requested. After the server receives requests from clients, it executes database retrieval , updates and manages data integrity and dispatches responses to client requests. The main aim of the server process is to perform the back-end tasks that are common to similar applications.

The message on the network has headers, these headers generate a considerable amount of overhead, look at fig.5-1. On wide-area networks, this overhead is not serious, where the number of bits/sec that can be send is typically fairly low. The limiting factor is the capacity of the lines, the CPUs are fast enough to keep the lines running at all speed. Probably a wide-area distributed system use the OSI or TCP/IP protocol without any loss in performance. On a LAN-based distributed system the overhead is serious. So much CPU time is wasted running protocols that the effective throughput over the LAN is often only a fraction of what the LAN can do.

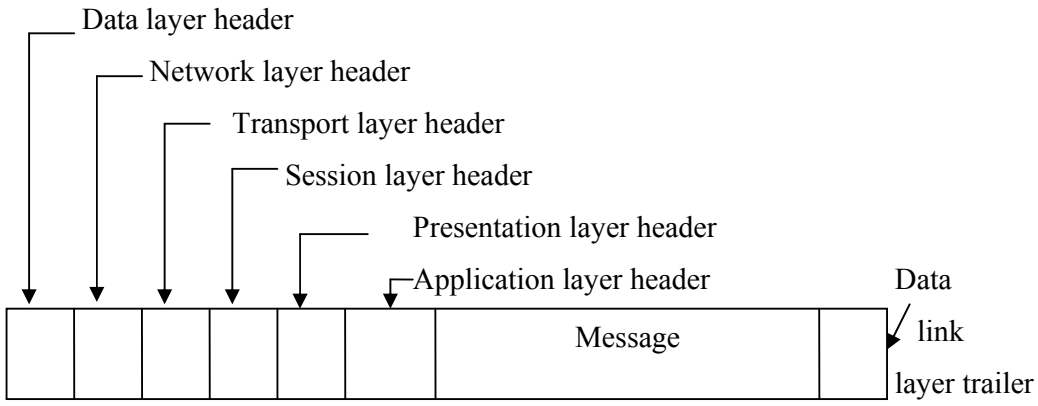


Fig.5-1. A typical message as it appears on the network.

5.2.Issues in Client-Server Communication

The design issues in this message passing system which concern the communication primitives are :

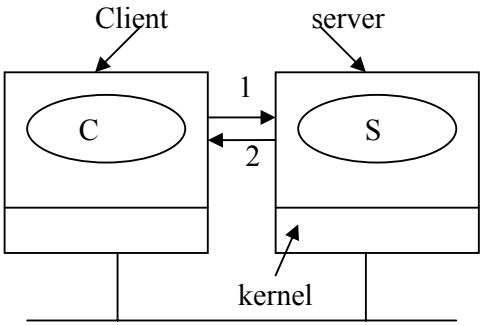
5.2.1.Addressing

Client must know server's address to send message. The file server has to be assigned a numerical address. If it refers to a specific machine, the sending kernel can extract it from the message structure and use it as the hardware address for sending the packet to the server. All the sending kernel has to do then is build a frame using the numerical address as the data link address and put the frame out on the LAN. The server's interface board will see the frame, recognize numerical address as it's own address, and accept it. The kernel will give the message to the only one process if there is one process running on the destination machine .If there are several processes running on the destination machine, consequently, a scheme that uses network address to identify a processes. Another method for addressing system is send messages to processes rather than to machine. Preferred method is to make address machine_number.process_number. This scheme is illustrated in fig.5-2.(a) no ambiguity between process 0 on machine 243 and process 0 on machine 199.

There are several methods for addressing processes, all with some problems:

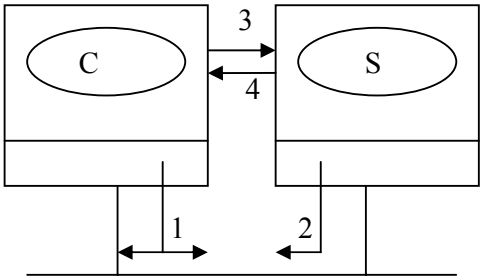
1. Hardwire machine address into client code. This is not transparent; user aware of where server is located, if replaced, all references must change.
2. Let each process pick its own identifier, then broadcast a locate packet. This places an extra load on the system as more packets are broadcast. look at fig .5-2.(b).

3. Name server maps ASCII service names to machine addresses, can be cached. Centralizing server addresses to one machine doesn't work in large systems. Name server can be replicated, but may have consistency problems. look at fig.5-2(c).
4. Another method uses special hardware, and processes pick random addresses. Network interface chips allow processes to store process addresses there. Frames use process addresses instead of machine addresses. Network inter-face chip examines frame to see if destination process is on its machine.



1:Request to 243.0
2:Reply to 199.0

(a)



1:Broadcast
2:Here I am
3:Request to 243.0
4:Reply to 199.0

(b)

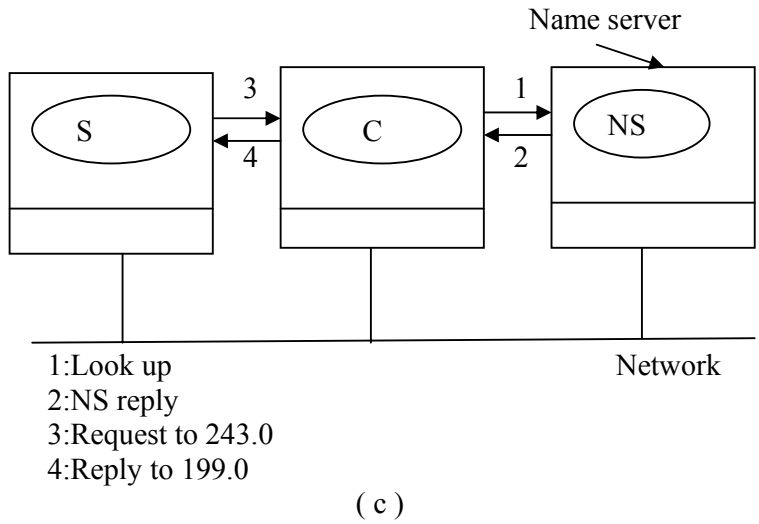


Fig.5-2.(a)Machine.process addressing.(b)Process addressing with broadcasting.(c)Address lookup via server .

5.2.2.Blocking versus Nonblocking Primitives

Blocking (or synchronous) message-passing primitives suspend during both the sending and receiving of a message, which might take hours, as shown in fig.5-3.(a) .

An alternative to blocking primitives are nonblocking primitives (or asynchronous) primitives returns caller control immediately.

The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission , instead of the CPU go idle .

The disadvantage with nonblocking primitives: cannot modify message buffer until message sent. Sender doesn't know when transmission completed so it can reuse the buffer.

One solution is to copy the message to an internal kernel buffer and go on, as shown in fig. 5-3.(b). When control is returned, buffer can be reused. Extra copy cuts efficiency.

Second solution is to interrupt sender when message has been sent. Disadvantage is that user-level interrupts lead to tricky programming and race conditions, farther more programs_based on interrupts are difficult to write correctly and nearly impossible to debug when they are wrong .

First choice is best under normal condition, but if overlapping processing and transmission is critical, nonblocking send with copying is best choice. Receive can be blocking or nonblocking. Wait, test and conditional_ receive primitives allow the receiver to

know when the operation has completed. As with send, the best choice is usually the blocking version of receive.

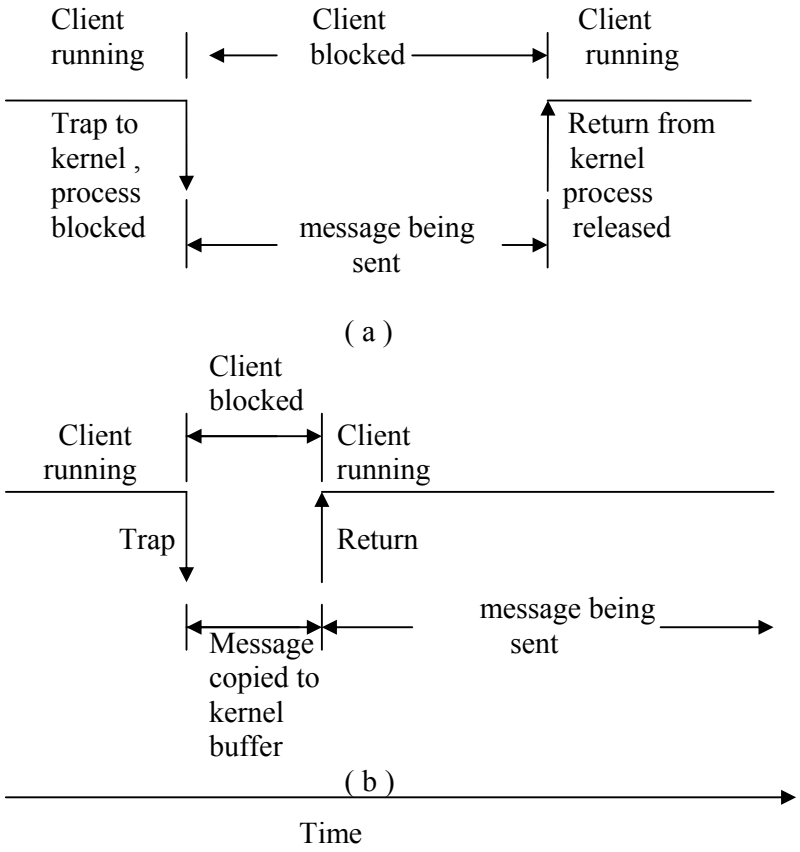


Fig.5-3. (a)A blocking send primitive. (b)A nonblocking send primitive**.**

5.2.3.Buffered versus Unbuffered Primitives

Primitives blow are unbuffered (address refers to a specific process). Caller is listening at a specific address and is prepared for one message, fig.5-4.(a) illustrates the use of an address to refer to a specific process. This works properly as server's receive come before client's send. If send is done first, how does server's kernel know where to copy message ?

1. Unbuffered: discard the message, let client time out and try again. If the client try again many consecutive attempts and fail, the client's kernel may give up, falsely concluding that the server has crashed or that the address is invalid.
2. Unbuffered: have receiving kernel keep messages around for awhile. The advantage of this method is reducing the chance that message will have to be thrown away. The disadvantage is of storing and managing prematurely arriving messages.

3. The buffers are needed and simple way of dealing with this buffer management is to define a new data structure called a mailbox. Mailbox might be created for the purpose of storing uncollected messages.

The mailbox technique is referred to as a buffered primitive. Problems arise when mailbox is full, deciding whether to keep or discard. May be solved by not letting a process send a message if no room to store. Sender must block until acknowledgement comes back saying message received, fig5-4.(b) illustrates the buffered primitive.

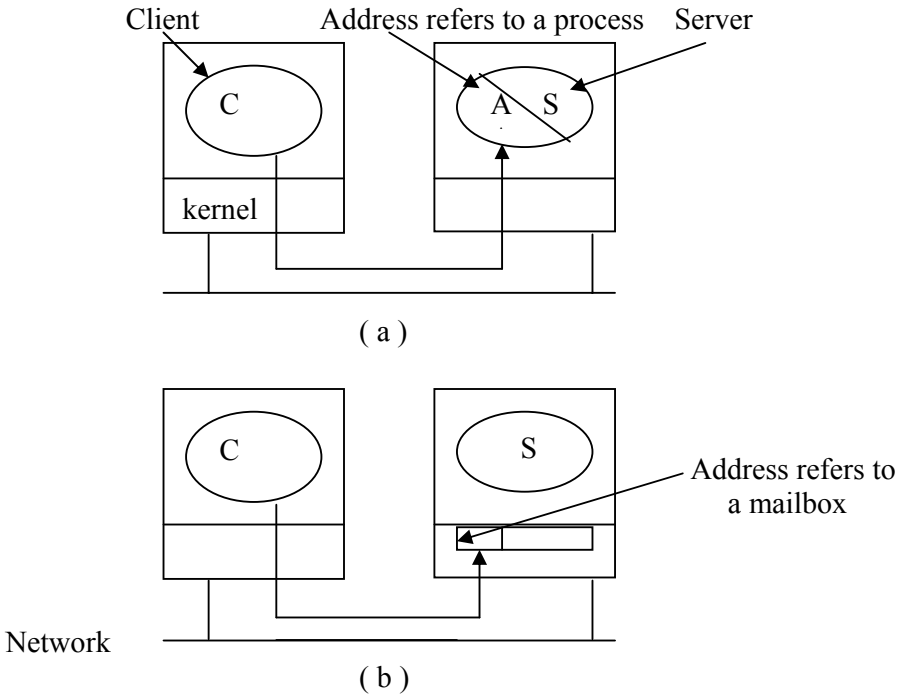


Fig.5-4.(a) Unbuffered message passing . (b) buffered message passing .

5.2.4.Reliable versus Unreliable Primitives

Assumption so for: messages sent will be received, but they can get lost .

There are three approaches to this problem :

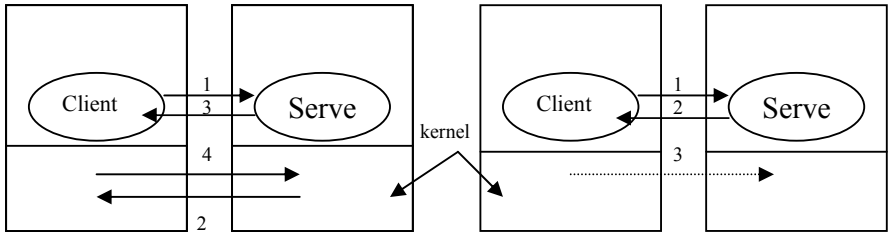
1. Redefine semantics of send to be unreliable .Assume no guaranties.
2. Redefine kernel on receiving machine to send acknowledgement to sending kernel. This required four messages to be send:

   The request and the acknowledgement of the request, the reply and the acknowledgement of the reply. This approach is shown in fig.5-5.(a).
3. In this approach the client is blocked after sending a message. The server's kernel does not send back an acknowledgement. Don't acknowledge kernel to kernel. Use

server to client reply as the implicit acknowledgement and only send client to server acknowledgement. This approach is shown in fig.5-5.(b).

The reason of an acknowledgement from client's kernel to the server's kernel requires extensive computation on the server.

A compromise between fig.5-5.(a) and fig.5-5.(b) that often works goes like this. When a request arrives at the server's kernel, a time is started .If the server sends the reply quickly enough, the reply functions as the acknowledgement .If the timer goes off, a separate acknowledgement is sent .Thus in most cases , only two messages are needed, but when a complicated request is being carried out, a third one is used.



1.Request (client to server)
2.ACK(kernel to kernel)
3.Reply (server to client)
4.ACK (kernel to kernel)

1.Request (client to server)
2.Reply (server to client)
3.ACK (kernel to kernel)

(a) (b)

Fig.5-5. (a) Individually acknowledged messages. (b) Reply being used as the acknowledgement of the request. Note the ACKs are handled entirely within the kernels.

5.2.5.Implementing the Client-Server Model

Fig.5-6. illustrated 81 different combinations of addressing, blocking, buffering and reliability. Most networks have maximum packet size, generally a few thousand bytes. Layer messages must be split up into multiple packets, may be lost, garbled.

One way of resolving issue of acknowledgements is that each packet is to be acknowledged, packet can be traced, but more traffic created .If each message acknowledged, fewer packets , but more complicated recovery.

Server packet types used in client-server protocol:

1. REQ Request packet from client to server says client wants service.
2. REP Request packet from server to client.
3. ACK Acknowledgement from either to other says previous packet arrived.

4. AYA Are you alive from client to server checks to see if server crashed.
5. IAA I am alive from server to client means server has not crashed.
6. TA Try again from server to client indicates server has no room.
7. AU Address unknown from server to client means no process is using address.

Item	Option 1	Option 2	Option 3
Addressing	Machine number	Sparse process addresses	ASCII names looked up via server
Blocking	Blocking primitives	Nonblocking with copy to kernel	Nonblocking with interrupt
Buffering	Unbuffered, discarding unexpected messages	Unbuffered temporarily keeping unexpected messages	Mailboxes
Reliability	Unreliable	Request-Ack-Reply Ack	Request-Reply-Ack

Fig.5-6. Four design issues for the communication primitives and some of the principal choices available.[2]

Chapter 6

Remote Procedure Call

Chapter 6

Remote Procedure Call

The most common communication protocol for communication between the clients of a service called self remote procedure call. The basic idea of RPC originated in work by Birrell and Nellson (1980). They suggested programs call procedures located on other machines. The calling process is usually suspended and the execution of the called procedure takes place in the other machine.

Although RPC sounds simple some problems exist:

1. Because procedures run on different machines they execute in different address spaces.
2. Parameters and results have to be passed , which can be complicated.
3. If both machines crash , each of the possible failures cause different problems.

6.1.How RPC works

The RPC tools make it appear to users as though a client directly calls a procedure located in a remote server program. The client and server each have their own address spaces; that is, each has it's own memory resource allocated to data used by the procedure. The following fig.6-1. illustrates the RPC architecture.

A remote procedure call occurs in the following steps:

- The client performs the following steps:
  1. The client procedure calls the client stub.
  2. The client stub builds a message and traps to the kernel.
  3. Call function in the kernel to sends the request and the message to the remote kernel.
- The server perform the following steps to call the remote procedure:
  1. The remote kernel accepts the request and give the message to the server stub.
  2. The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs.
  3. The server stub calls the actual procedure on the server.
- The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is complete, a similar sequence of steps returns the data to the client:
  1. The server (remote procedure) does the work and returns the result to the stub.

2. The server stub converts output parameters to the format required for transmission over the network (pack it in a message) and trap to the kernel.
3. The remote kernel sends the message to the client's kernel.

- The client completes the process by accepting the date over the network and returning it to the calling function:
  1. The client kernel gives the message to the client stub.
  2. The client stub converts the data to the format used by the client computer. The stub writes data into the client memory (unpack the result) and returns the result to the calling program on the client.

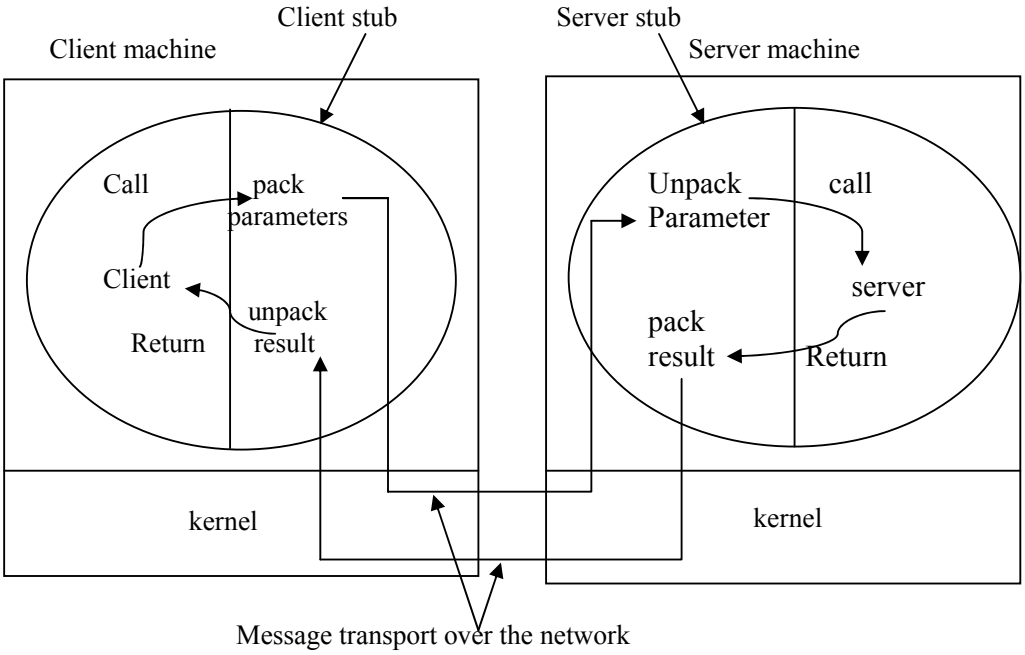


Fig.6-1. Calls and messages in an RPC. Each ellipse represents a single process, with the portion being the stub.[7]

6.2.Marshaling and Data types

The client stub packs the parameters into a message and sends them to the server stub. Packing parameters into a message is called parameter marshaling.

The server stub after examining the message that it receives from the client stub calls the appropriate server.

The purpose of a data marshaling mechanism is to represent the caller's argument in away that can be efficiently interpreted by a server program. However, in a large distributed system it is common that multiple machine types are present. A condition that complicate parameter passing, for example, non – identical client/server machines:

- First problem, IBM mainframes use the EBCDIC character cod, whereas IBM personal computers use the ASCII.
- Second problem is that integer representations vary for the most common CPU chips. On some chips the most significant byte of an integer is also the low byte of the first word in memory. While on others the most significant bytes is stored in the high byte of the last word of the integer. These are called little-endian and big-endian representations. This causes some data to be read incorrectly.[6]

6.3.Finding the Server

We look at two ways to do this.

- First method, hard code server network address. This is done at compile time when the client remote procedure call is being built. The problem is that this method is extremely inflexible, since you can only change the remote server address if you recompile.
- Second method, dynamic binding, to avoid all the problem in the first method, some distributed systems use what is called dynamic binding to match up client and servers. The client stub is given the locations of the different servers at execution time of the RPC when it reaches the client stub.

To achieve this, we must have a formal server specification and a binder program. The binder program, which runs on the client, registers available servers. Its list of available servers and the services they offer is made available to the client stub.

Server formal specification contains the following information:

- Name of the server (and network location).
- version number of the procedures.
- list of procedures provided by the server.

The primary use of the formal specification is as input to the stub generator. It is the stub generator that creates the client and server stubs. When the server begins executing, it registers itself by sending a message to the binder program on client computer. The message to the binder contains the server's name, version number, unique identifier, and server network address (handle to locate it).The handle is system dependent. It could be an Ethernet address, an IP address, a process identifier, or something else. Server can also send a

deregistration message to a client binder program to remove offered services. The binder interface is shown in fig.6-2.

Call	Input	Output
Register	Name, version, handle, unique id	
Deregister	Name, version, unique id	
Look up	Name, version	Handle, unique id

Fig.6-2. The binder interface

When a client executes an RPC call for the first time. The client stub contacts the binder program asking for a procedure matching the client's request. The binder program checks for a match. If not match is found, the request fails. If a match is found, the binder program gives the handle of the server, as the server network address, to which the RPC call is to be sent. The client stub uses the handle as an address to send the request message to. The message contains the unique identifier, which the server's kernel uses to redirect the message.

The advantages of dynamic binding is:

- Can handle multiple servers with the same interface through the server's unique identifier.
- Binder program can spread client requests over multiple servers (load balancing).
- Binder programs can poll servers periodically, deregistering any server that does not respond.
- Binder programs can assist with authentication (verifying servers/clients).
- Binder programs can verify that the client and server are using the same version of a procedure(of the interface).

The disadvantages of dynamic binding is:

- Lots of overhead in importing and exporting interfaces (especially since many client processes are short lived and start all over again).

- In a large distributed system, you may need multiple binders and they must stay synchronized.[2]

6.4.RPC Semantics in the Presence of Failures

6.4.1.Client Cannot Locate the Server

Two Possible ways to deal with this type of failure:

1. Each of the procedures returns a value, with the code-1 used to indicate failure, and set a global variable..

   Problem: the solution is not general enough.

2. Have the client stub raise an exception or send a signal (e.g. SIGNOSERVER), causing the exception or signal handler to be invoked.

   Problem:

   - Not every language supports exceptions or signals .
   - Having to write the exception or signal handler destroys the transparency goal of RPC .

6.4.2.Lost Request Messages

Have kernel start a timer when sending the request .If the timer expires before a reply come back, kernel sends the request again.

Problem: Kernel could falsely conclude that the server is down.

6.4.3.Lost Reply Messages

Some operation can be safely repeated with no damage being done. A request that has this property is said to be **idempotent .**

We can't just resent the message from the server and get away with it. For example, in a monetary transaction, the request is to take $20 out of our account. Since the request got processed by the server, if we just resent the message from the client, we will ask for another $20 to be removed from the account. Not a good plan because some operations are **not idempotent**. We must identify resend messages by using sequence

numbers so that the server can throw away the resend if it has already executed a previous request.

6.4.4.Server Crashes

possibilities of server crash as shown in fig.6-3.

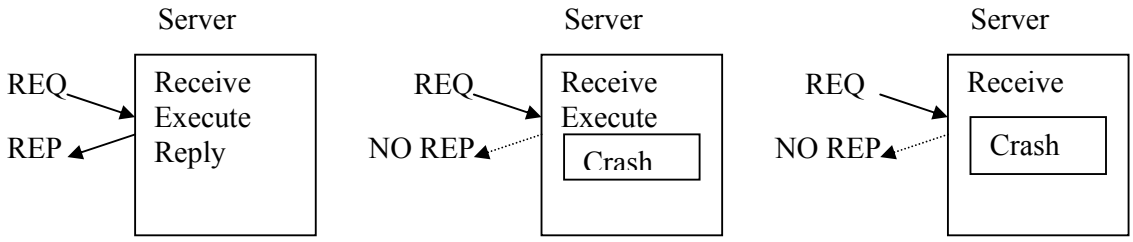


Fig.6-3.(a) Normal case. (b) Crash after execution. (c) Crash before execution.

Three ways to deal with server crashes:

- At least once semantics: the client waits until the server reboots retries the operation. The goal is to keep trying until you get a reply.
- At most once semantics: Give up immediately and report failure. The RPC can only have been done one time or not at all.
- Use no semantics (do nothing): When the server crashes the client gets no help and no promises.

6.4.5.Client Crashes

We will assume that the client sends a request to a server and crashes before receiving a reply from the server. This causes Orphan Computations to be performed by the server. An orphan computation is a computation that is active with no one waiting for the results.

Problems with Orphans Computation is:

- Wasting CPU time and tying up other valuable resources .
- Confusion can result when the rebooted client receives replies of previous requests.[7]

6.5.Implementation Issues for RPC System , with a special emphasis on the performance

6.5.1.Acknowledgements

Sometime the clients and server in an RPC environment need to exchange a large block of data, there are many strategies to do this:

The first is stop-and-wait protocol: client breaks down it's request into packets, sends each packet, and waits for an acknowledgement for each packet that is sent.

- Advantage : If a packet is lost only the lost packet needs to be resent.
- Disadvantage : Each packet must be acknowledged.

The alternative is blast protocol: Client sends all of its packets and only waits for an acknowledgement for all of its packets .

- Advantage : Only one acknowledgement.
- Disadvantage : A lost packet requires all packets to be resent.

The alternative often called selective repeat can buffer all packet comes in correctly, hope the other packet comes in correctly and then specifically ask the client to send it the bad packet. Flow control techniques are needed to govern the amount of information being sent by one processor and received by another processor (and vice versa). If not carefully managed, overrun errors can occur because the buffers on the network cards overflow. With stop-and-wait overrun errors are impossible because the second packet is not sent until the receiver has explicitly indicated that it is ready for it. Blast protocol is more efficient than stop-and-wait. However, there are also ways of dealing with overrun, because a smart sender can insert a delay between packets to give the receiver just enough time to generate the packet-arrived interrupt and reset itself. That if the problem is caused by the chip being disable temporarily while it is processing an interrupt.

If the problem is caused by the finite buffer capacity of the network chip, say n packets, the sender can send n packets followed by a substantial gab.

6.5.2.Critical path

The critical path is the sequence of instruction that is executed on every RPC, and is depicted in fig.6-4.

The steps of critical path from client to server is after the client stub has been called, its first job is to prepare message buffer into which it can assemble the outgoing message. When the underlying packet format has a substantial number of fields that must be filled in, this method is especially appropriate, but does not change from call to call. The second job

marshal parameters into buffer along with the rest of the header fields then the message is ready for transmission , so the trap to the kernel is issued . Next, the context switch to kernel, then the kernel must copy the message into its address space so it can access it, determine destination address, put address in message header, set up network interface and finally start timer. On the server machine, by the receiving hardware the bits will come in and be put either in an on-board buffer or in memory. After that the receiver process interrupt. Check packet for validity by the interrupt handler, and decide which stub to give it to. If stub is waiting then the message is copied to the stub. The context switch to server stub is done. The server unmarshals the parameters, setup parameters on stack and then call the server when every thing is ready.

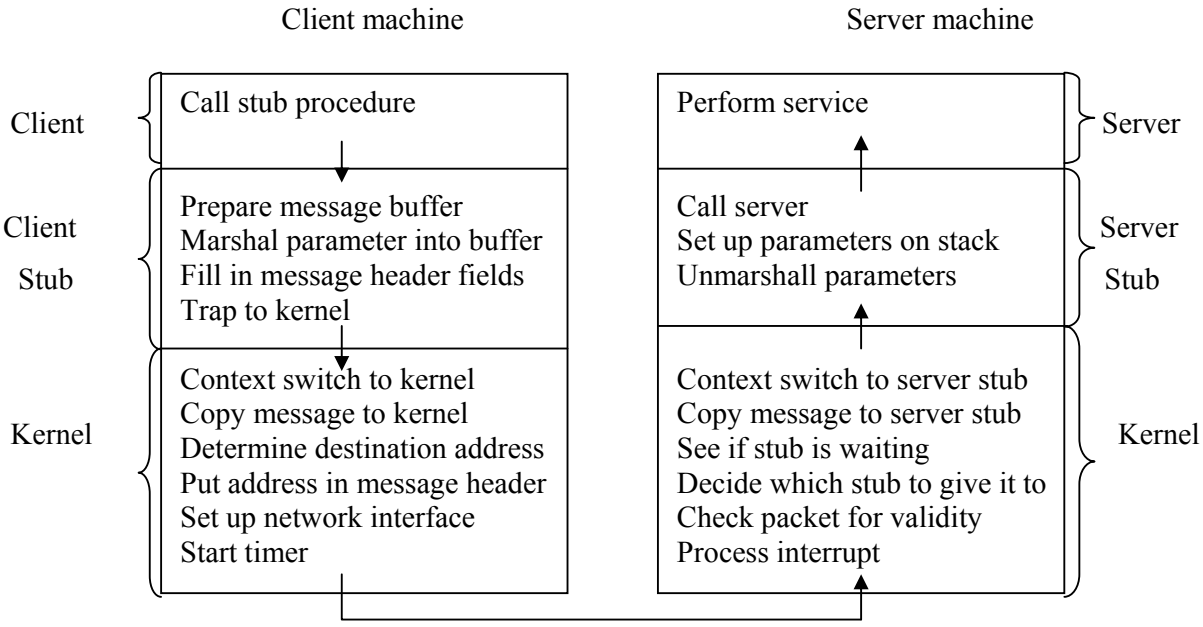


Fig.6-4. Critical path from client to server

6.5.3.Timer Management

Both the client and servers use timer (usually through the communication device derivers)to ensure proper message delivery. The timer expires and the original message is retransmitted, if the reply is not forthcoming with the expected time. This process is repeated until the sender gets bored and gives up. Managing and keeping track of timers for each message sent over the network is complex.

6.5.4.Copying

The Copying dominates RPC execution time. In most systems the number of times a message must be copied depends on hardware, software, and type of call. In the best case, just one copy is required, where as in the worst case eight copies are required.

Scatter-gather is a feature of the hardware that greatly helps eliminate unnecessary copying. A network chip that can do scatter-gather can be set up to assemble a packet by concatenating two or more memory buffers.[2]

Chapter 7

Group Communication

Chapter 7

Group Communication

With group communications we desire to have a single message sent to multiple processes (possibly running on multiple processors).With RPC a single message, multiple times, to multiple receivers. Group communications are different than RPC based communications. Specifically: Group can be organized in multiple ways and group can be addressed in many ways.

7.1.Group Communications-the Basic

Mechanisms are needed for managing groups and group membership. A group is dynamic, new groups can be created and old ones destroyed on the fly. A process enters a group by requesting membership in the group, and leaves by announcing that it is leaving the group. A process can be a member of multiple groups at the same time. A group allows a process to deal with a collection of processes as a single abstraction. A process sends message to a group without knowing how many processes are in the group and the network address of the processes in the group. On some networks it is possible to send a message to a special address where multiple processes can listen. When a packet is sent to one of these addresses all listing processes receive the message. This technique is known as multicasting. Implementing group communications with multicasting is easy, it only requires just sending a message to a multicast address. If multicasting is not available, some networks support broadcasting. With broadcasting a message is sent to all processes on the network. Processes interested in broadcasted messages, accept the message. broadcasting is not as efficient as multicasting because individual processes must filter messages that they are interested in and many messages are sent in a broadcast that can saturate the network. If multicasting or broadcasting is not available on the network, a machine can sent multiple messages to all of the members in the group. This is known as unicasting. Unicasting requires multiple packets (one for each group member), multicasting and broadcasting requires a single packet that is sent to multiple receivers.

7.2.Design Issues for Group Communications

Group communications share many design considerations with point-to-point communications such as blocking versus nonblocking, buffered versus nonbuffered. Group communications also have many additional options that may be utilized.

7.2.1.Closed Groups versus Open Groups

There are two categories that support group communication

1. Closed groups:Only members of the groups can communicate with the group. Closed groups typically used for parallel processing.

2. Open groups: Any process can communicate with the group. Fig.7-1.show the difference between closed and open groups.

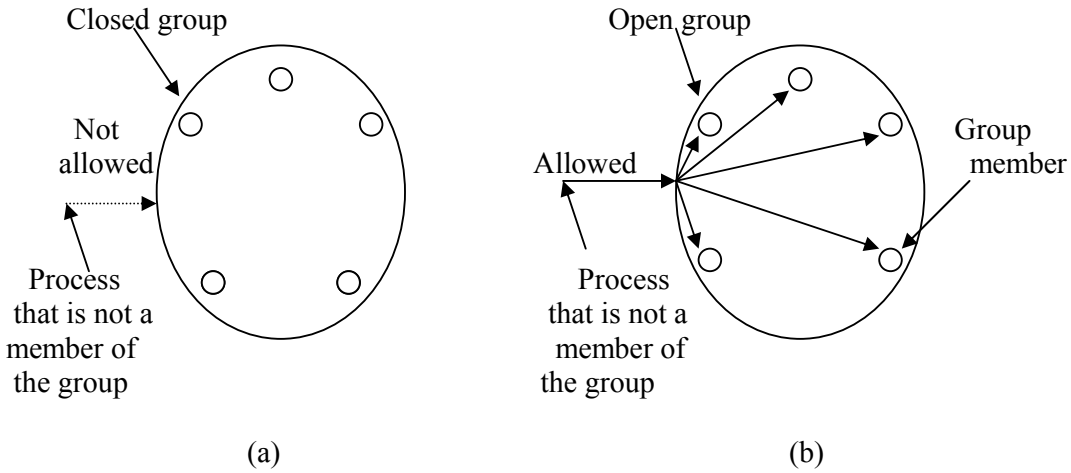


Fig.7-1.(a) Outsiders may not send to a closed group.(b) Outsiders may send to an open group.

7.2.2.Peer Group versus Hierarchical Group

Peer groups versus hierarchical groups with the internal structure of the group.

1. In peer groups no central group coordinator exists all process are in the same level decisions are made collectively.
2. In hierarchical Groups a central coordinator for the group dispatches messages to all group member.

Fig.7-2. illustrated communication in a peer group and in a simple hierarchical group.

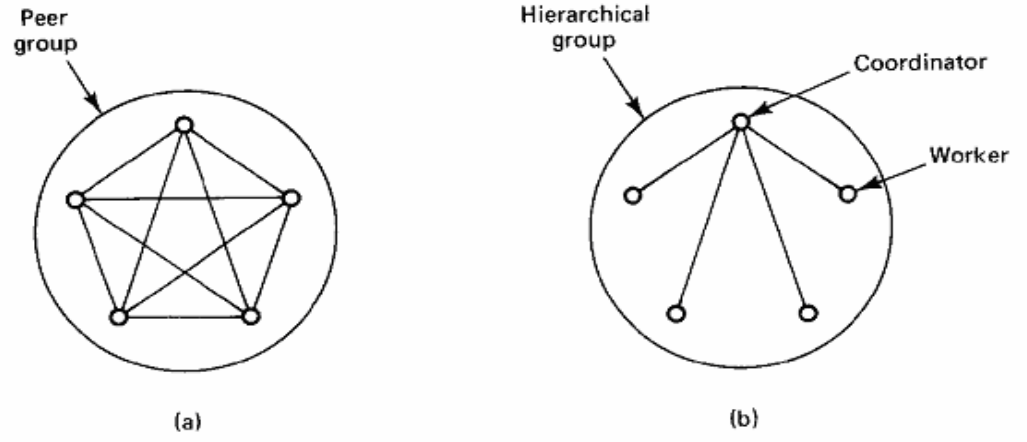


Fig.7-2. (a) Communication in a peer group. (b) Communication in a simple hierarchical group.

The advantages of the peer groups is the symmetric and the system has no single point of failure. In case of a process crash, the group simply becomes smaller, but still can continue. The disadvantage of this organization is that decision making is complicated and voting has to be taken for decision making thus the organization involves some delay and overhead.

The hierarchical group has no symmetric and the system has single point of failure. Loss of the coordinator brings the entire group to a dead stop, but the system can make decisions without bothering everyone else.

7.2.3.Group Membership

Group communication requires some method for creating and detecting groups and also for allowing processes to enter and exit a group. One way of handling this is to use a group server to coordinate all groups in the distributed system. The group server maintains the entire data base and membership of all the groups. This approach suffers from the disadvantage that if the group server crashes the group management no longer exist and possibly all the groups have to be reconstructed. The second approach is to manage a group membership in a distributed way. Process can send a message to all group members announcing its intention to enter/exit the group. There are two issues to be considered here. First one, if a process crashes all group members will still think the crashed process is still a member of the group and individual group members will individually have to discover the crashed process. The second issues for synchronization of the group, as soon as a process enters the group it must start receiving all group messages and as soon as a process leaves the group it must no longer receive group messages. A final issue, in case many machines went. Some protocol is needed to rebuild the group. If two or three processes try to take the initiative at the same time the protocol will have to be able to withstand.

7.2.4.Group Addressing

For a process to send a message to a group there should be some means of specifying which group to be addressed. The first method of addressing is to give each group a unique address. If network allows multicasting , then sending is straightforward. Otherwise, if network supports broadcasting, then kernel has to filter messages and each broadcast contains the group address. Kernel examines each broadcast message and extracts the group address and forwards the message to any local processes belonging to the group. If network only supports unicast, the sending processes kernel has to send a message to each other member of the group. Fig.7-3. below show the three implementation methods. A second

method of group addressing requires the sender to have a detailed list of all destinations (e.g IP addresses). The parameter in the call to send that specifies the destination is a pointer to a list of addresses. This method is not transparent because processes must be aware of who is a member of which group. Whenever group membership changes, process must update their membership lists.

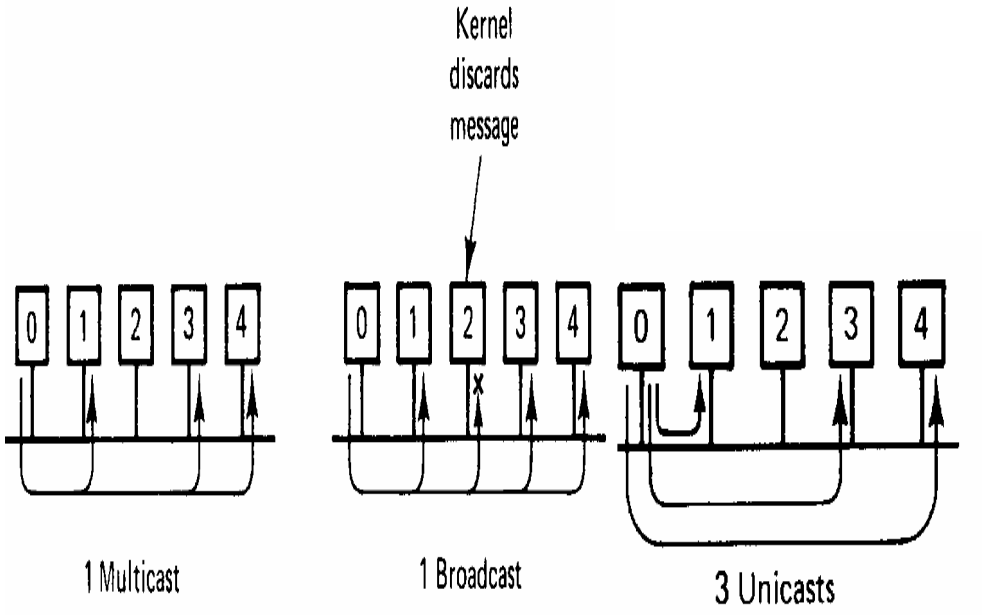


Fig.7-3. Process 0 sending to a group consisting of processes 1,3, and 4, (a) Multicast implementation.(b) Broadcast implementation. (c) Unicasts implementation.

Predicate addressing allows the sender to add a boolean condition to each message delivered to a member of the group only if the condition to each message. A message is delivered to a member of the group only if the condition is satisfied.

7.2.5.Atomicity

The group communication environment requires the all-or-nothing property, hard to implement because unreliability of network, varying buffer spaces, machine failures, process crashes, etc. (even network support multicasting). In case one member of this group can not receive a message to. we allow the other group members to receive the message. If this is permitted, we require atomicity or atomic broadcast (all-or-nothing delivery). Atomic broadcast is desirable because it makes programming distributed systems much easier. Atomicity prevents inconsistency. If a message is received by a member of the group it is

received correctly by other members. Atomicity is highly desirable in fault-tolerant systems where consistency across distributed processes is a requirement.

An alternative algorithm is the sender sends message to all members of the group, timers are set and retransmission sent as necessary . When a process receives a new message, message is sent to all other members of the group, again, process starts timers and retransmits as necessary. This method guarantees that all surviving members of the group will eventually receive the message.

7.2.6.Message Ordering

Consistent message ordering is desirable to make group communications easy to understand. When multiple messages are sent to the group, all group members should see the messages in the same order. This property is easy to comprehend if multicasting or broadcasting is used and difficult to realize if unicasting is used to implement group communications. Look to fig .7-4.

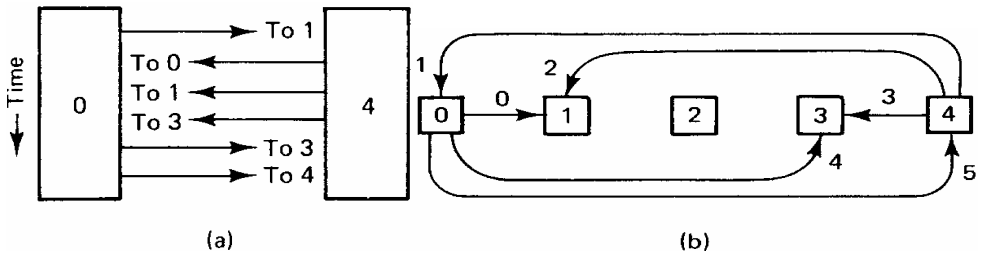


Fig.7-4.(a) The three messages sent by processes 0 and 4 are interleaved in time. (b) Graphical representation of the six messages, showing the arrival order.

Message ordering types:

- Global time ordering:

  Global time ordering is based on all messages being received by all processes in exactly the order the messages were sent. Message ordering very difficult to realize in the absence of global time. However global time is difficult (or impossible) to synchronize to a high degree of accuracy between distributed processes.

- Consistent time ordering:

  Consistent time ordering is weaker time ordering than global time ordering. With consistent time ordering all processes receive all messages in the same order. The order in which the messages are received may not be the exact order in which the message were sent.

7.2.7.Overlapping Groups

A process can be a member of multiple groups at the same time, but if global or consistent time ordering is used in groups, this ordering may not extend to Overlapping Groups (Fig.7-5.).

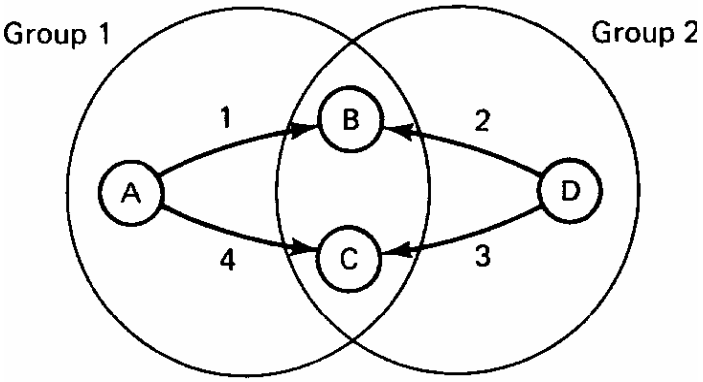


Fig.7-5. Four processes, A, B, C, and D, and four messages. Processes B and C get the messages from A and D in a different order.

In the above figure it is possible for processes B and C see messages sent from processes A and D in different order even though a time ordering mechanism is used in group 1 and group 2.[2]

Chapter 8
Design for Communication in Distributed Systems

Chapter 8

Design for Communication in Distributed Systems

8.1. Client-Server model programming

In this program use socket for interprocess communication. Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person.

The client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel.[5]. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the `socket()` system call.
2. Connect the socket to the address of the server using the `connect()` system call.
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the `socket()` system call .
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call.
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
5. Send and receive data.

8.2.Socket Types

When a socket is created, the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used address domains, the unix domain, in which two processes which share a common file system communicate, and the Internet domain, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format.

The address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32 bit address, often referred to as its IP address). In addition, each socket needs a port number on that host.

There are two widely used socket types, stream sockets, and datagram sockets. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communications protocol. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.[8]

8.3.The C code for client server communication

C codes for a very simple client and server are provided in appendix (A). These communicate using stream sockets in the Internet domain.

When you run the server, you need to pass the port number in as an argument. You can choose any number between 2000 and 65535. If this port is already in use on that machine, the server will tell you this and exit. If this happens, just choose another port and try again. If the port is available, the server will block until it receives a connection from the client.

To run the client you need to pass in two arguments, the name of the host on which the server is running and the port number on which the server is listening for connections.

The client will prompt you to enter a message. If everything works correctly, the server will display your message on stdout, send an acknowledgement message to the client and terminate. The client will print the acknowledgement message from the server and then terminate. The flow chart (Fig.8-1.) illustrates the scenario of the client code:

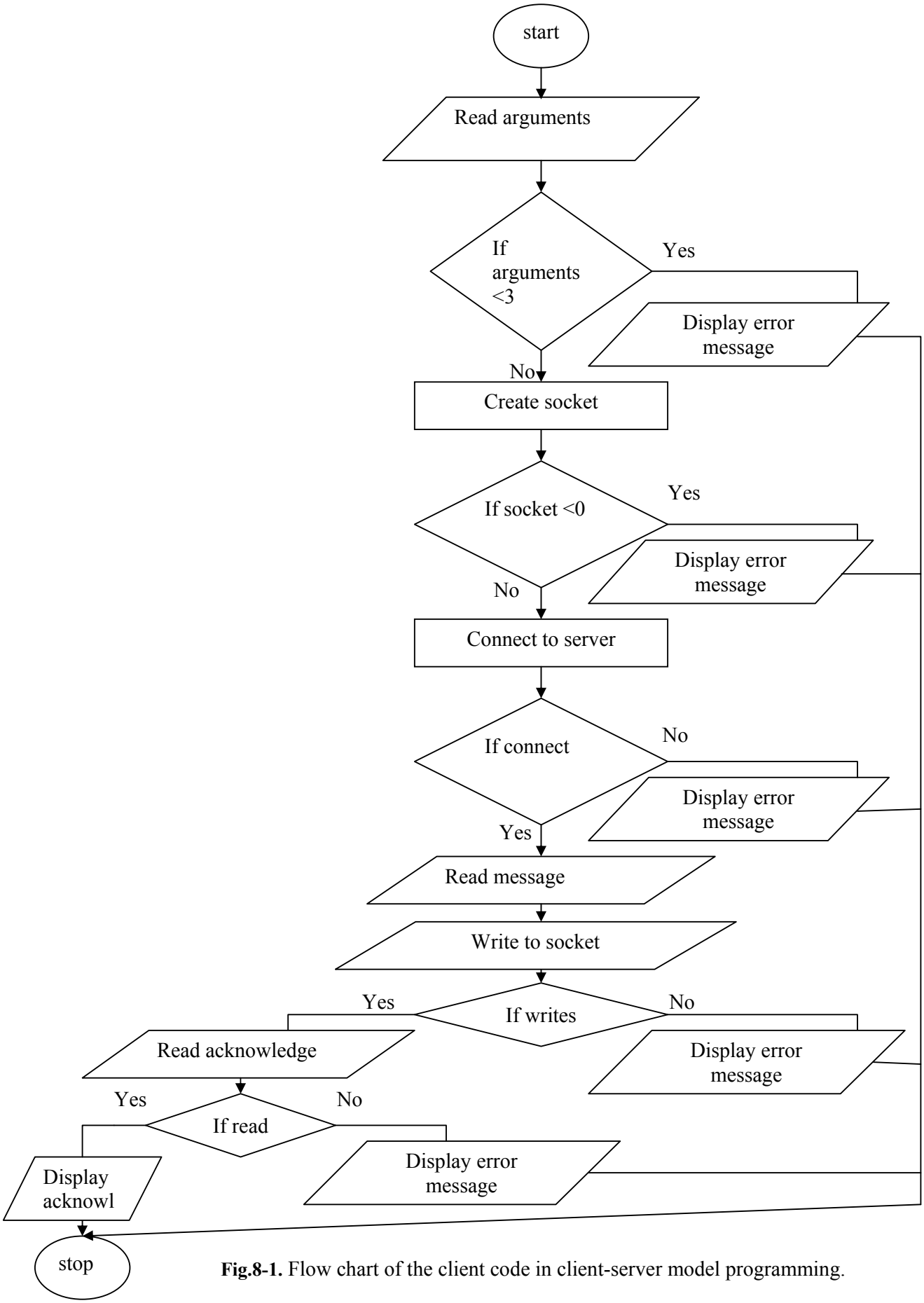
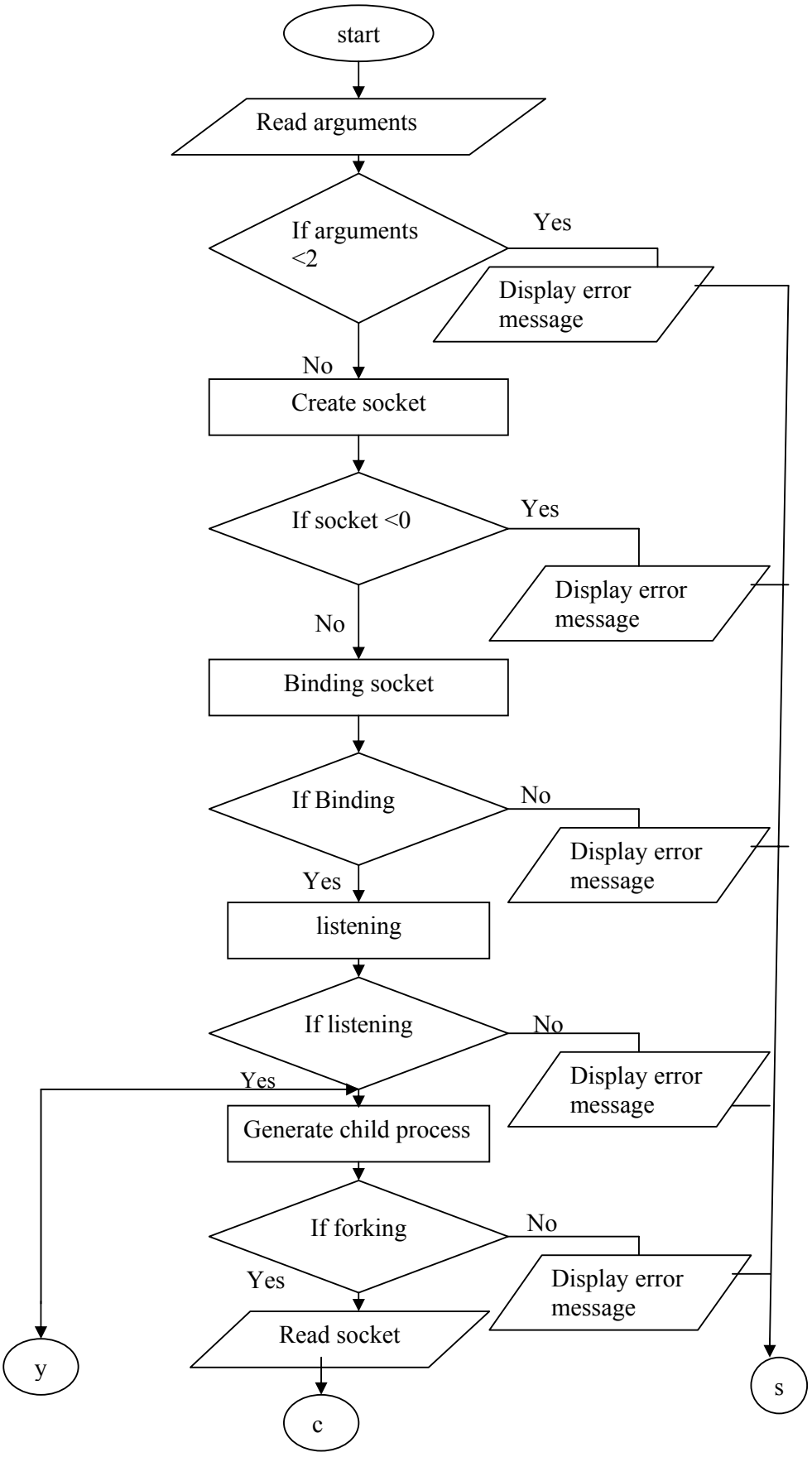


Fig.8-1. Flow chart of the client code in client-server model programming.

The flow chart(Fig.8-2.) bellow illustrates the scenario of the server code:



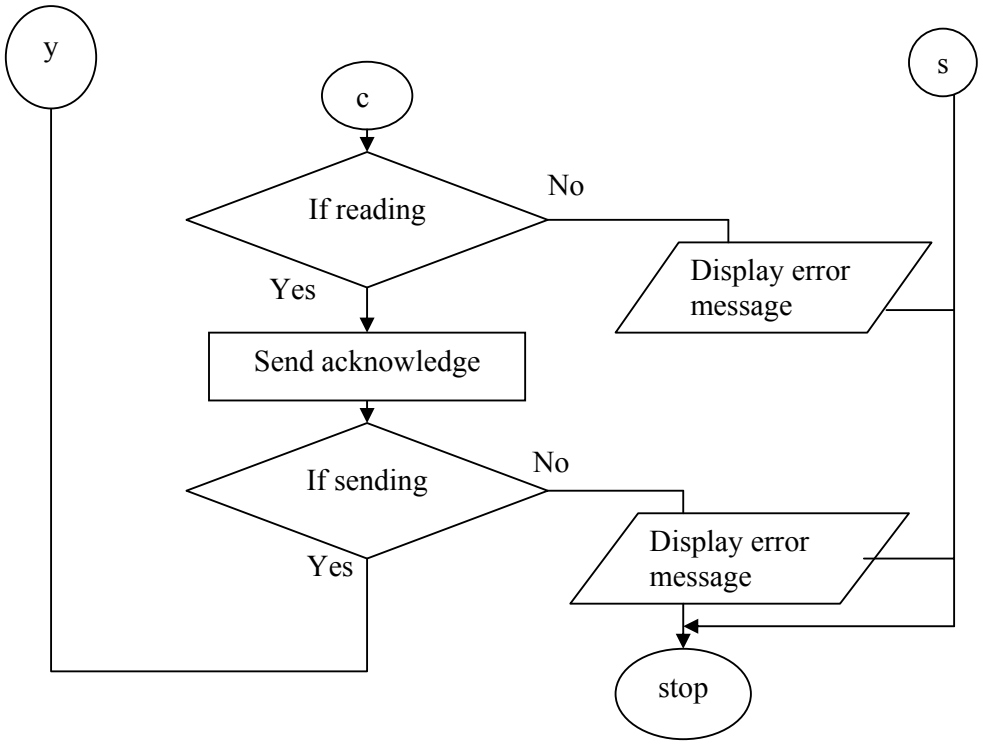
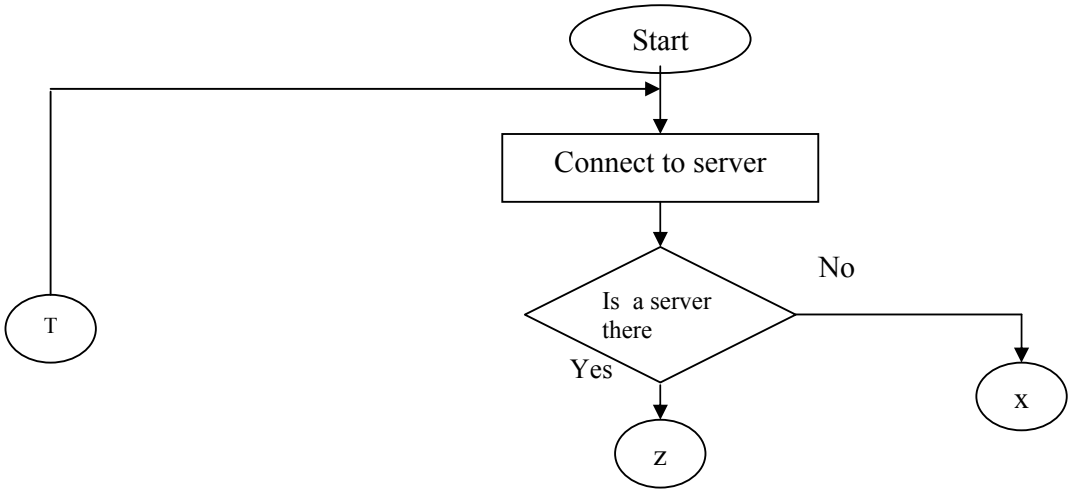


Fig.8-2. Flow chart of the server code in client-server model programming.

8.4.Remote Procedure Call programming

In this program use Open Networking Computing (ONC) PRC for client server interaction. The purpose of the program is to create a server that implements functions to add/subtract two integers and return the result. C codes for a very simple interface file, client and server are provided in appendix (B).

- The interface file is mc.x.
- The client code is in the file mc.c.
- The server code is in the file mc_svc_proc.c.
- The flow chart (Fig8-3.) illustrates the scenario of the client code.



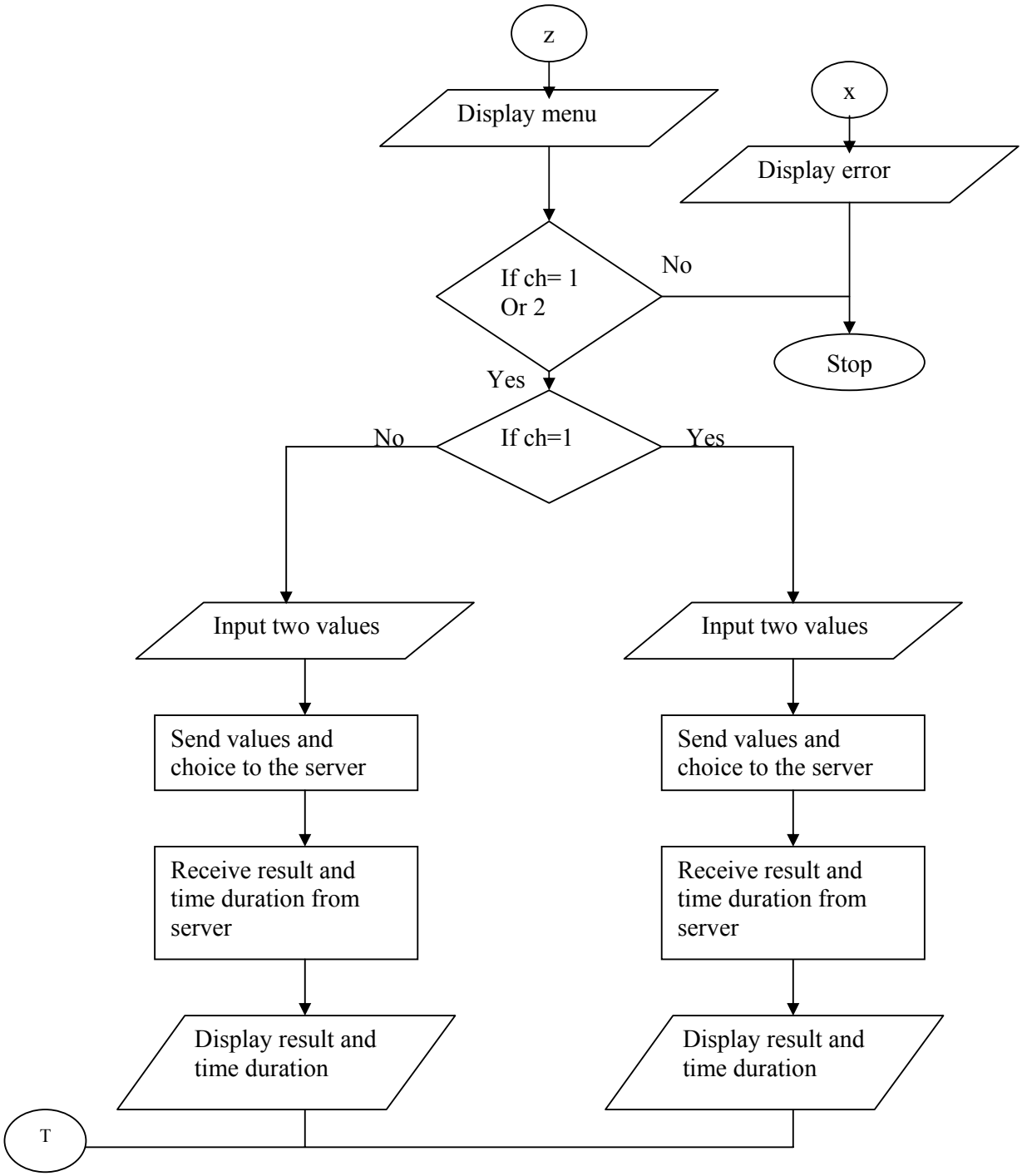


Fig.8-3. flow chart of the client code in remote procedur call programing.

The flow chart (Fig.8-4.) bellow illustrates the scenario of the server code:

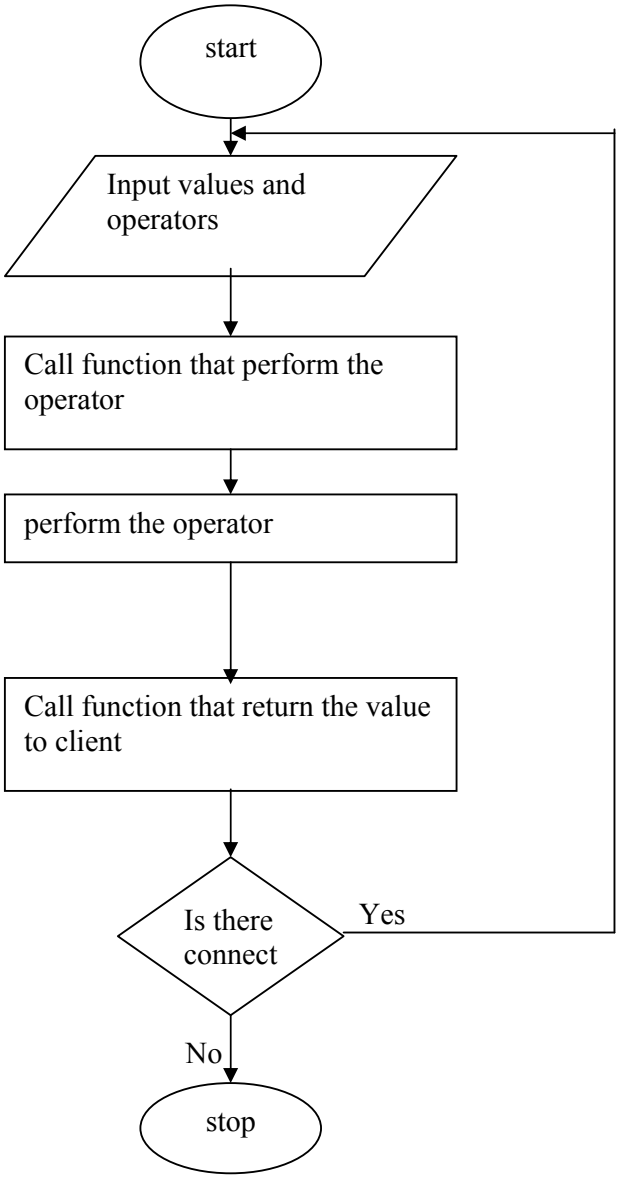


Fig.8-4. flow chart of the server code in remote procedur call programing.

8.5.Group Communication programming

In this program use multicast for group interaction. The purpose of this program is to create a multicastsender that send data to group of multicastsniffer. Java codes for multicast sender and multicastsniffer in appendix (C). The flow chart (Fig8-5.) bellow illustrates the scenario of the multicastsniffer code:

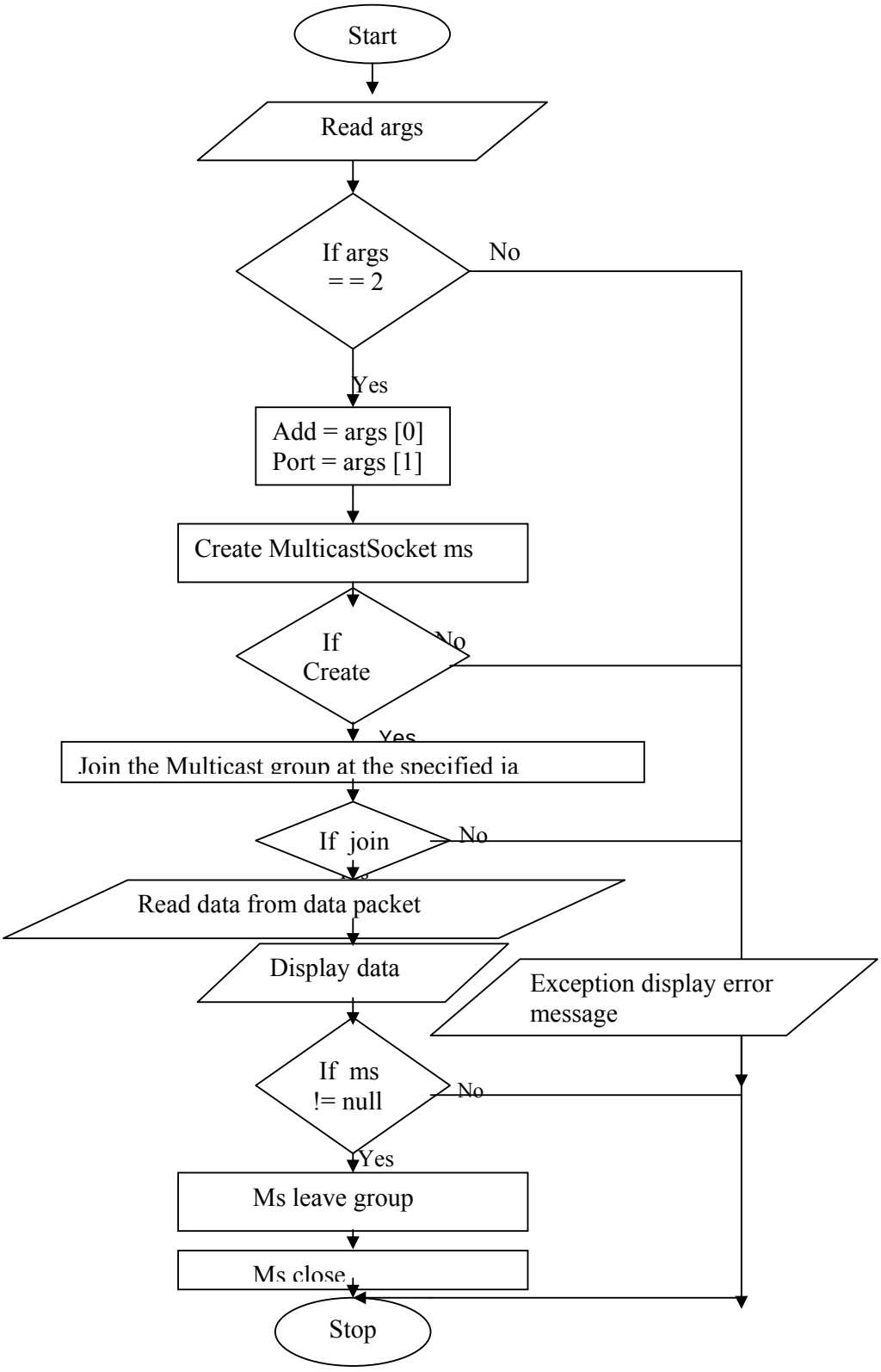
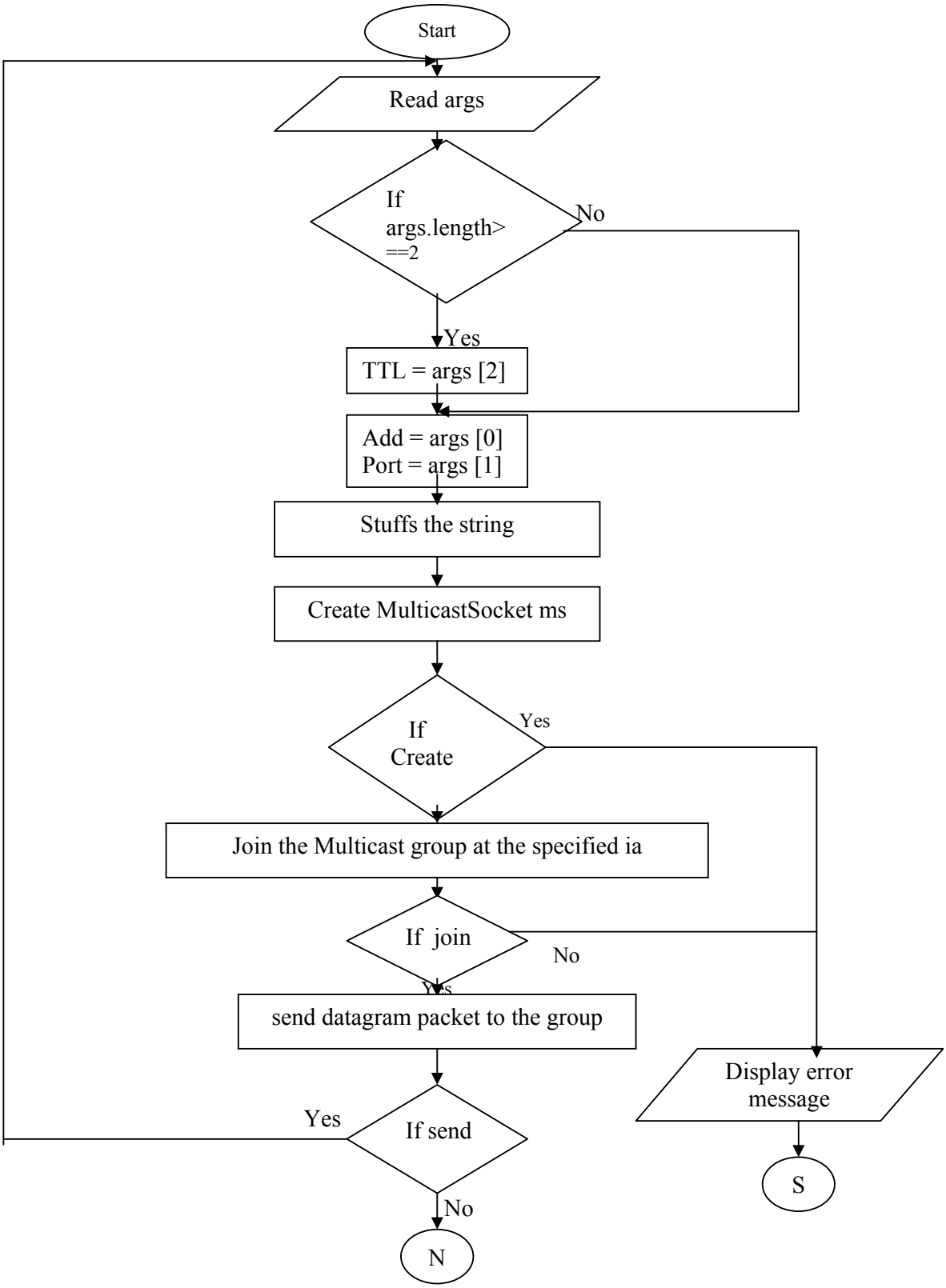


Fig.8-5. flow chart of the Multicastsniffer code in group communication programming.

The flow chart (Fig.8-6.) bellow illustrates the scenario of the Multicastsender code:



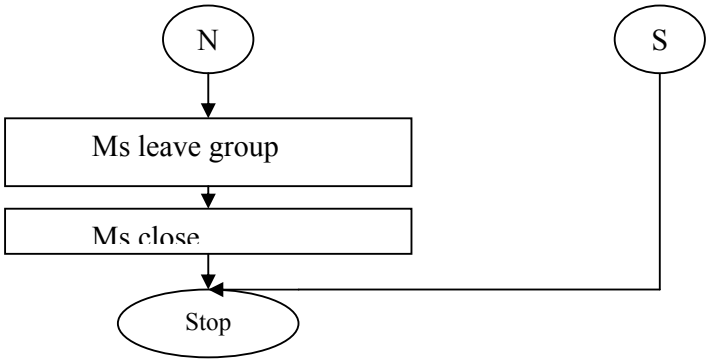


Fig.8-6. flow chart of the Multicastsender code in group communication programming.

8.6. Communicating with a Multicast Group

Once a MulticastSocket has been created, it can perform four key operations. These are:

1. Join a multicast group.
2. Send data to the members of the group.
3. Receive data from the group.
4. Leave the multicast group.

8.7.Results

The results achieved from running of the program are discussed underneath.

8.7.1.Client-Server model programming

First run the server program by enter the port number and client program by enter the name of the host and the port number. After that the client will prompt you to enter the message, once you enter the message the message displays in the server. The server sends the acknowledgement to the client. The client will then print the acknowledgement message from the server. Figs (8-7) and (8-8) illustrate the result of communication between client and server.

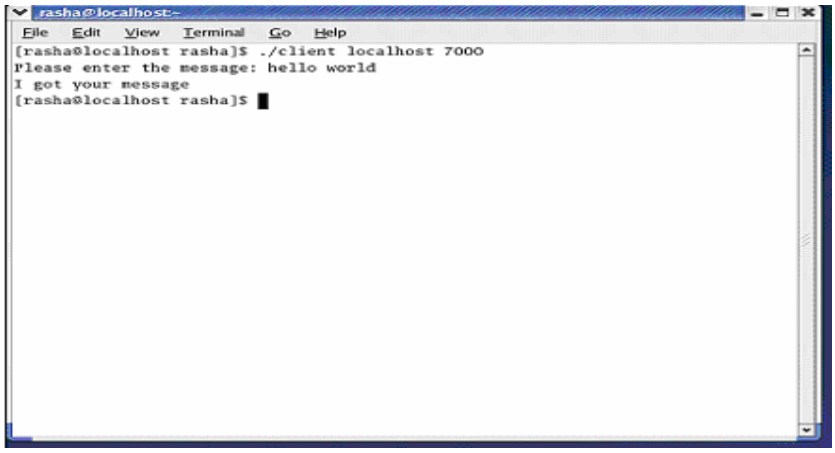


Fig.8-7. Client Result

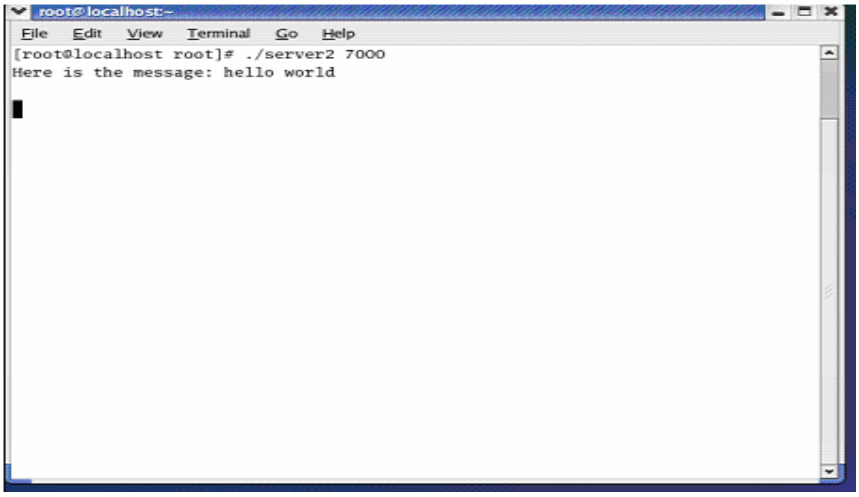


Fig.8-8. Server Result

8.7.2.Remote Procedure Call programming

First run the server program (fig.8-11) and client program. Following that the client will prompt you to enter the number of operation. Once entered the client will prompt you once again to enter two integers. When entered, the client will print the result and the duration time as illustrated in figs(8-9) and (8-10).

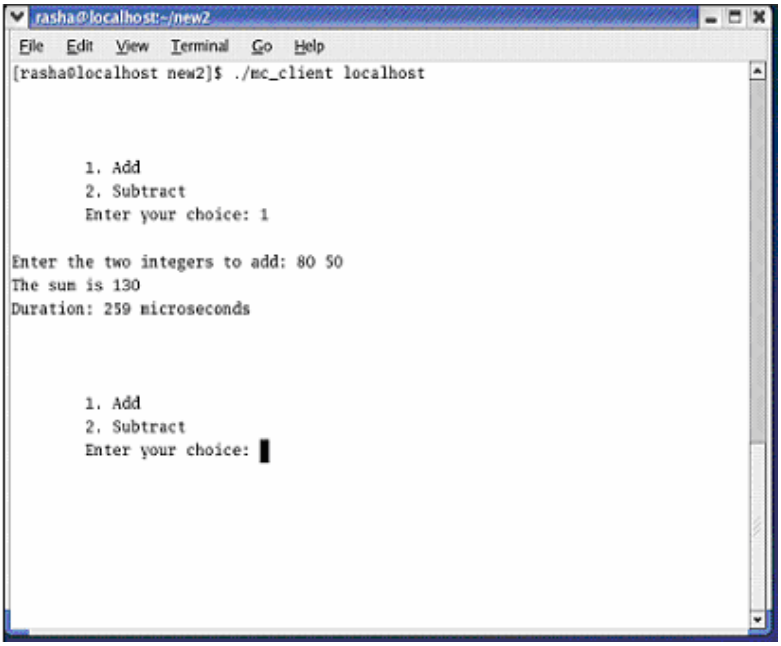


Fig.8-9. Add Result(Client Result)

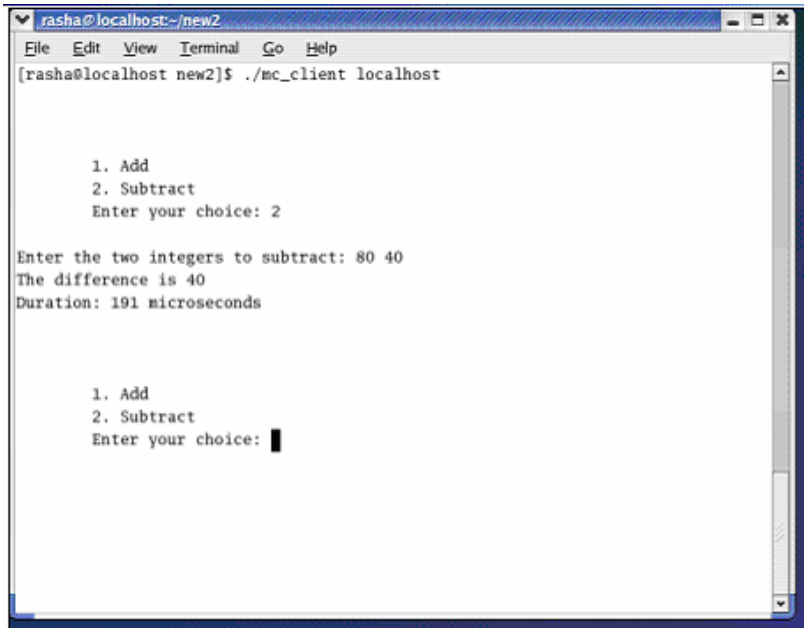


Fig.8-10. Subtract Result(Client Result)

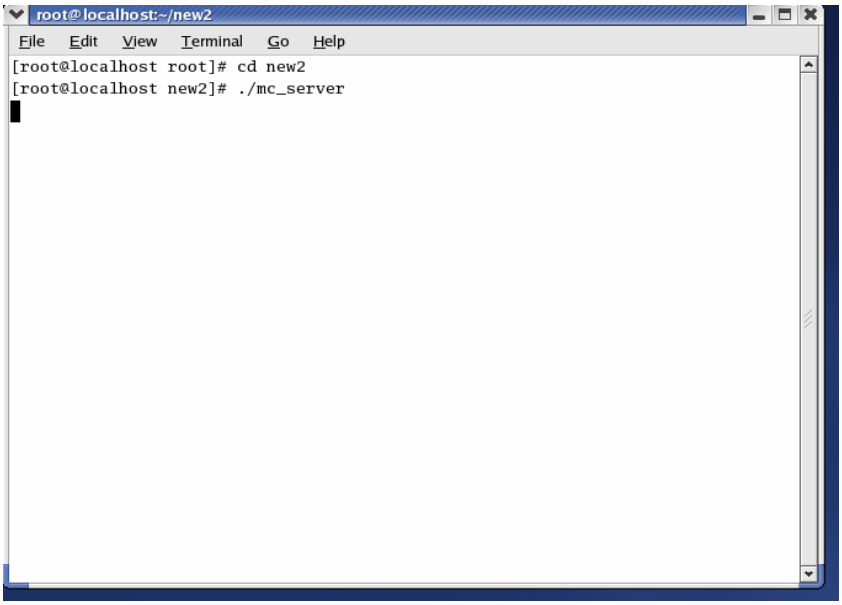


Fig. 8-11. Server Result

8.7.3.Group Communication programming

First run the multicastsniffer. Then send data to that group by running multicastsender. Figs (8-12) and (8-13) illustrate the result of communication with multicast group.

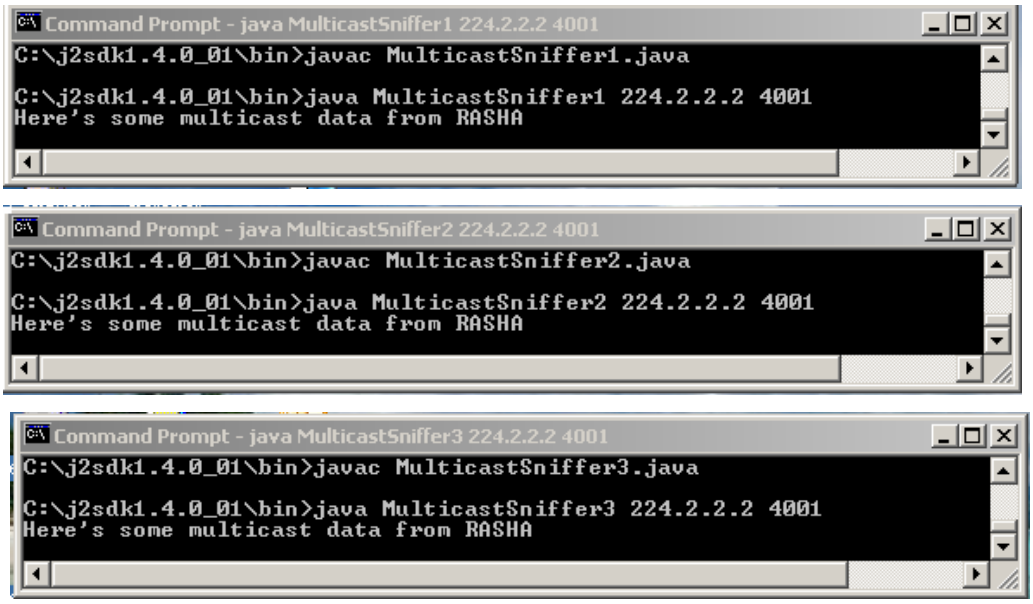


Fig. 8-12. Multicastsniffe Result

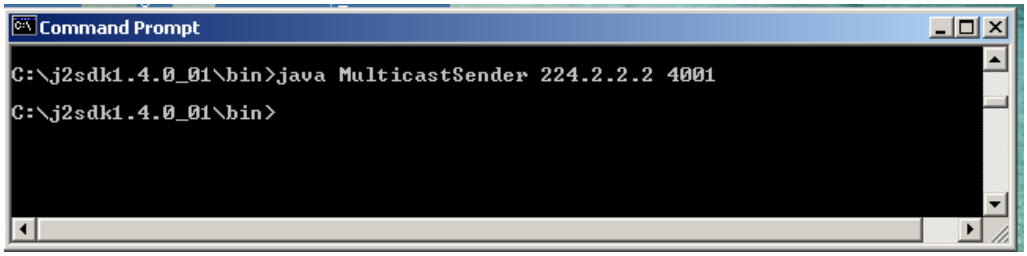


Fig. 8-13. Multicastsender Result

Chapter 9

Analysis of the program

Chapter 9

Analysis of the program

9.1. Client-Server model

Server code

The following system calls that I have used in my server code to establishing a socket:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

The socket() system call creates a new socket. It takes three arguments. The first is the address domain of the socket. The symbol constant AF_UNIX is used for the unix domains, and AF_INET for the Internet domain.

The second argument is the type of socket. Stream socket (SOCK_STREAM) in which characters are read in a continuous stream, and a datagram socket (SOCK_DGRAM), in which messages are read in chunks. The third argument is the protocol. If this argument is zero (and it always should be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose TCP for stream sockets and UDP for datagram sockets.

The socket system call returns an entry into the file descriptor table (i.e. a small integer). This value is used for all subsequent references to this socket. If the socket call fails, it returns -1. In this case the program displays and error message and exits.

```
if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
            error("ERROR on binding");
```

The `bind()` system call binds a socket to an address, in this case the address of the current host and port number on which the server will run. It takes three arguments, the socket file descriptor, the address to which is bound, and the size of the address to which it is bound. The second argument is a pointer to a structure of type `sockaddr`, but what is passed in is a structure of type `sockaddr_in`, and so this must be cast to the correct type. This can fail if the socket on this machine is already in use.

```
listen(sockfd,5);
```

The `listen` system call allows the process to listen on the socket for connections. The first argument is the socket file descriptor, and the second is the size of the backlog queue, i.e., the

number of connections that can be waiting while the process is handling a particular connection. This should be set to 5, the maximum size permitted by most systems. If the first argument is a valid socket, this call cannot fail, and so the code doesn't check for errors.

```
while (1) {
    newsockfd = accept(sockfd,
          (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
       error("ERROR on accept");
    pid = fork();
    if (pid < 0)
       error("ERROR on fork");
    if (pid == 0)  {
       close(sockfd);
       dostuff(newsockfd);
       exit(0);
    }
    else close(newsockfd);
 } /* end of while */
```

The `accept()` system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. The second argument is a reference pointer to the address of the client on the other end of the connection, and the third argument is the size of this structure.

The dostuff(int sockfd)is a dummy function this function will handle the connection after it has been established and provide whatever services the client requests. Once a connection is established, both ends can use read and write to send information to the other end, and the details of the information passed back and forth do not concern us here.

To allow the server to handle multiple simultaneous connections, we make the following:

1. Put the accept statement and the following code in an infinite loop.
2. After a connection is established, call fork() to create a new process.
3. The child process will close sockfd and call dostuff, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by dostuff() returning, this process simply exits.
4. The parent process closes newsockfd. Because all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection.

```
bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n",buffer);
```

Would only get to this point after a client has successfully connected to our server. This code initializes the buffer using the `bzero()` function, and then reads from the socket. The `read()` will block until there is something for it to read in the socket, i.e. after the client has executed a `write()`. It will read either the total number of characters in the socket or 255, whichever is less, and return the number of characters read.

```
n = write(newsockfd,"I got your message",18);
if (n < 0) error("ERROR writing to socket");
```

Once a connection has been established, both ends can both read and write to the connection. Naturally, everything written by the client will be read by the server, and everything written by the server will be read by the client. This code simply writes a short message to the client. The last argument of write is the size of the message.

Client code

The following system calls that I have used in my `client` code to establishing a socket:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

This code is the same as that in the server.

```
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
```

The `connect` function is called by the client to establish a connection to the server. It takes three arguments, the socket file descriptor, the address of the host to which it wants to connect (including the port number), and the size of this address. This function returns 0 on success and -1 if it fails.

```
   printf("Please enter the message: ");
   bzero(buffer,256);
   fgets(buffer,255,stdin);
   n = write(sockfd,buffer,strlen(buffer));
   if (n < 0)
       error("ERROR writing to socket");
   bzero(buffer,256);
   n = read(sockfd,buffer,255);
   if (n < 0)
       error("ERROR reading from socket");
   printf("%s\n",buffer);
   return 0;
   }
```

This code prompts the user to enter a message, uses fgets to read the message from stdin, writes the message to the socket, reads the reply from the socket, and displays this reply on the screen

9.2. Remote Procedure Call

mc.x

This code defines the definitions of the functions to be called remotely. The individual functions are identified as 1 (for add) and 2 (for subtract).

mc.c

The client code is written with knowledge of the information specified in the interface and with knowledge of three RPC library functions, **clnt_create**, **clnt_pcreateerror**, and **clnt_destroy** but not of socket commands like **socket**, **bind**, etc.

mc_svc_proc.c

This code is written with knowledge of the information specified in the interface but without need of socket and network commands or of RPC library functions. It consists just of the functions that will be called remotely and of auxiliary data structures and definitions.

Rpcgen

Rpcgen is a protocol compiler. The compilation of the flowing:

- mc.c RPC Client – written by programmer
- mc_svc_proc.c RPC Server-written by programmer
- mc.x Interface definition-written programmer

generates:

- mc.h Header file-generated by rpcgen
- mc_xdr.c Interface funvtion-generated by rpcgen
- mc_clnt.c Client-generated by rpcgen
- mc_svc.c Server-generated by rpcgen

- a header file mc.h which expresses the interface information and must be included by both the client, mc.c, and the server, mc_svc_proc.c
- a c file, mc_clnt.c, that will be linked with the client. It contains calls to the ONC function **clnt_call** which performs the calls to the remote procedures and interfaces with the XDR representation standard. The actual calls for interfacing with XDR are in the next file, mc_xdr.c. The combination of mc_clnt.c and mc_xdr.c is called the **Client Stub**.
- a c file, mc_xdr.c, that will be linked with both the client and the server, and contains code for placing/ removing information from the network messages in accordance to the XDR format.
- a c file, mc_svc.c, that represents the main program of the server and will be linked with the server file mc_svc_proc.c and with the mc_xdr.c file. In mc_svc_proc.c we find used the following functions from ONC and XDR APIs **pmap_unset, svcudp_create, svc_register, svctcp_create, svc_run, svc_sendreply, svcerr_noproc, svc_getargs, svcerr_decode, svc_freeargs, svcerr_systemerr**.

9.3. Group Communication

Multicast Sniffer program

The program begins by reading the name and port of the multicast group from the first command-line argument. Next, we create a new MulticastSocket ms on the specified port. This socket joins the multicast group at the specified InetAddress. Then it enters a loop in which it waits for packets to arrive. As each packet arrives, the program reads its data, converts the data to an ISO Latin-1 String, and prints it on System.out. Finally, when the user interrupts the program or an exception is thrown, the socket leaves the group and closes itself.

Multicast Sender program

The program begins by reading the address of a multicast group, a port number, and an optional TTL from the command line. It then stuffs the string "Here's some multicast data\r\n" into the byte array data using the getBytes( ) method, and places this array in the DatagramPacket dp. Next, it constructs the MulticastSocket ms, which joins the group ia. Once it has joined the group, ms sends the datagram packet dp to the group ia . The TTL value is set to one to make sure that this data doesn't go beyond the local subnet. Having sent the data, ms leaves the group and closes itself.

Chapter10

Conclusions and Recommendations

Chapter10

Conclusions and Recommendations

10.1.Conclusions

- Layered Protocols (OSI, TCP/IP) is well suited for wide-area networking.
- Client-Server is simplest and most widely adopted programming model for distributed systems . One entity provides a service; another entity uses the service. This allows them to be placed in different computers, have different implementations, different OS and hardware.
- TCP/IP socket programming is the essential tool for developing client/server software.
- Socket programming is the basis for most distributed systems, also sockets are more efficient, so socket is major in networking programming.
- Two interprocess communication techniques: message passing and remote procedure call (RPC).
- Remote procedure calls are a high-level communication paradigm that allows programmers to write network applications using procedure calls that hide the details of the underlying network. RPC is a powerful technique for constructing distributed, client/server based applications without requiring that callers be aware of the underlying network.
- A programming language which uses RPC must have some means of compiling, binding, and loading distributed programs onto the network.
- The remote procedure call mechanism uses several of low level connection routines. These routines are generated automatically by a compiler, called rpcgen. We will generate two RPC programs using rpcgen.
- Group Communication is used for one to many communications in distributed systems.
- By providing no response to a delivered request message the group communication facility is only able to provide unreliable group communication.
- UDP protocol must be used for broadcast or multicast applications.

10.2.Recommendations

The following development to improve the performance of group communication is recommend:

- Interaction between the group should be developed to monitor communication within the group in such a manner that disclose the identity of the sender.

- A novel method of addressing called predicate addressing can be adopted. Each message sent to members contains a predicate to be evaluated. The predicate can include machine number. If the predicate evaluates true the message is accepted. If not it is rejected.

Appendices

Appendices

Appendix (A)

The Client-Server Communication program

Server code

```
        /* A simple server in the internet domain using TCP
   The port number is passed as an argument
   This version runs forever, forking off a separate
   process for each connection
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void dostuff(int); /* function prototype */
void error(char *msg)
{
    perror(msg);
    exit(1);
}
int main(int argc, char *argv[ ])
{
     int sockfd, newsockfd, portno, clilen, pid;
     struct sockaddr_in serv_addr, cli_addr;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
```

```
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
            sizeof(serv_addr)) < 0)
            error("ERROR on binding");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    while (1) {
       newsockfd = accept(sockfd,
             (struct sockaddr *) &cli_addr, &clilen);
       if (newsockfd < 0)
          error("ERROR on accept");
       pid = fork();
       if (pid < 0)
          error("ERROR on fork");
       if (pid == 0)  {
          close(sockfd);
          dostuff(newsockfd);
          exit(0);
       }
       else close(newsockfd);
    } /* end of while */
    return 0; /* we never get here */
}
/******** DOSTUFF() *********************
 There is a separate instance of this function
 for each connection.  It handles all communication
 once a connection has been established.
 *****************************************/
void dostuff (int sock)
{
  int n;
```

```
   char buffer[256];

   bzero(buffer,256);
   n = read(sock,buffer,255);
   if (n < 0) error("ERROR reading from socket");
   printf("Here is the message: %s\n",buffer);
   n = write(sock,"I got your message",18);
   if (n < 0) error("ERROR writing to socket");
}
```

Client code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
void error(char *msg)
{
   perror(msg);
   exit(0);
}
int main(int argc, char *argv[])
{
   int sockfd, portno, n;
   struct sockaddr_in serv_addr;
   struct hostent *server;
   char buffer[256];
   if (argc < 3) {
      fprintf(stderr,"usage %s hostname port\n", argv[0]);
      exit(0);
   }
   portno = atoi(argv[2]);
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
       error("ERROR opening socket");
```

```
server = gethostbyname(argv[1]);
if (server == NULL) {
   fprintf(stderr,"ERROR, no such host\n");
   exit(0);
}
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
    (char *)&serv_addr.sin_addr.s_addr,
    server->h_length);
serv_addr.sin_port = htons(portno);
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
   error("ERROR connecting");
printf("Please enter the message: ");
bzero(buffer,256);
fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
if (n < 0)
    error("ERROR writing to socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n < 0)
    error("ERROR reading from socket");
printf("%s\n",buffer);
return 0;
```

Appendix (B)

Remote Procedure Call programming

The interface file

```
/*
 * mc.x: remote calculator access protocol
 */
struct mypair{
  int arg1;
  int arg2;
};
/* program definition */
program MCPROG { /* could manage multiple servers */
        version MCVERS {
                int ADD(mypair) = 1;
                int SUBTRACT(mypair) = 2;
        } = 1;
} = 0x20000002;  /* program number ranges established by ONC */
```

Server code

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "mc.h"
int *
add_1_svc(mypair *argp, struct svc_req *rqstp)
{
        static int  result;
        /* Start of inserted code */
        result = argp->arg1 + argp->arg2;
        /* End of inserted code */
        return &result;
}
```

```
int *
subtract_1_svc(mypair *argp, struct svc_req *rqstp)
{
        static int  result;
        /* Start of inserted code */
        result = argp->arg1 - argp->arg2;
        /* End of inserted code */
        return &result;
}
```

Client code

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "mc.h"
void
mcprog_1(char *host)
{
        CLIENT *clnt;
        int  *result_1;
        mypair  add_1_arg;
        int  *result_2;
        mypair  subtract_1_arg;
#ifndef  DEBUG
        clnt = clnt_create (host, MCPROG, MCVERS, "tcp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif          /* DEBUG */
        /* Start of inserted code */
         while (1) {
```

```
char choice[256];
struct timeval before, after;
long dure;
printf("\n\n");
printf("\n\t1. Add\n");
printf("\t2. Subtract\n");
printf("\tEnter your choice: ");
scanf("%0255s", choice);
if (strcmp(choice, "1") == 0) {
  printf("\nEnter the two integers to add: ");
  scanf("%d %d", &(add_1_arg.arg1), &(add_1_arg.arg2));
  /* Start of code generated by rpcgen */
  gettimeofday(&before, NULL);
  result_1 = add_1(&add_1_arg, clnt);
  gettimeofday(&after, NULL);
  if (result_1 == (int *) NULL) {
          clnt_perror (clnt, "call failed");
  }
  /* End of code generated by rpcgen */
  printf("The sum is %d\n", *result_1);
}else if (strcmp(choice, "2") == 0) {
  printf("\nEnter the two integers to subtract: ");
  scanf("%d %d", &(subtract_1_arg.arg1), &(subtract_1_arg.arg2));
  /* Start of code generated by rpcgen */
  gettimeofday(&before, NULL);
  result_2 = subtract_1(&subtract_1_arg, clnt);
  gettimeofday(&after, NULL);
  if (result_2 == (int *) NULL) {
           clnt_perror (clnt, "call failed");
  }
  printf("The difference is %d\n", *result_2);
  /* End of code generated by rpcgen */
} else
  break;
```

```
            dure = (after.tv_sec - before.tv_sec)*1000000 +
                                (after.tv_usec -before.tv_usec);
            printf("Duration: %ld microseconds\n", dure);
        }
        /* End of inserted code */
#ifndef  DEBUG
        clnt_destroy (clnt);
#endif   /* DEBUG */
}
int
main (int argc, char *argv[])
{
        char *host;
        if (argc < 2) {
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        mcprog_1 (host);
exit (0);
}
```

Appendix (C)

Multicast Sender

```
import java.net.*;
import java.io.*;
public class MulticastSender {
public static void main(String[] args) {
InetAddress ia = null;
int port = 0;
byte ttl = (byte) 1;
// read the address from the command line
try {
ia = InetAddress.getByName(args[0]);
port = Integer.parseInt(args[1]);
if (args.length > 2) ttl = (byte) Integer.parseInt(args[2]);
}
catch (Exception e) {
System.err.println(e);
System.err.println(
"Usage: java MulticastSender multicast_address port ttl");
System.exit(1);
}
byte[] data = "Here's some multicast data from RASHA\r\n".getBytes( );
DatagramPacket dp = new DatagramPacket(data, data.length, ia,
port);
try {
MulticastSocket ms = new MulticastSocket( );
ms.joinGroup(ia);
for (int i = 1; i < 2; i++) {
ms.send(dp, ttl);
}
ms.leaveGroup(ia);
ms.close( );
}
```

```
catch (SocketException se) {
System.err.println(se);
}
catch (IOException ie) {
System.err.println(ie);
}}}
```

Multicast Sniffer

```
import java.net.*;
import java.io.*;
public class MulticastSniffer {
public static void main(String[] args) {
InetAddress group = null;
int port = 0;
// read the address from the command line
try {
group = InetAddress.getByName(args[0]);
port = Integer.parseInt(args[1]);
} // end try
catch (Exception e) {
// ArrayIndexOutOfBoundsException, NumberFormatException,
// or UnknownHostException
System.err.println(
"Usage: java MulticastSniffer multicast_address port");
System.exit(1);
}
MulticastSocket ms = null;
try {
ms = new MulticastSocket(port);
ms.joinGroup(group);
byte[] buffer = new byte[8192];
while (true) {
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
ms.receive(dp);
String s = new String(dp.getData( ));
```

```
System.out.println(s);
}}catch (IOException e) {
System.err.println(e);
}finally {
if (ms != null) {
try {
ms.leaveGroup(group);
ms.close( );
}catch (IOException e) {}
}}}}
```

References

[1] A.S.Tenebaum (1996), Computer Networks, Prentice-Hall International, Inc.

[2] Andrew Tanenbaum (1995), Modern Operating systems, Prentice Hall.

[3]G.Coulouris, J.Dollimore, T.Kindberg 2001, Distributed system Concepts and Design, Pearson Education Limited.

[4]MCSE Networking Essen Plus (2000) pub. By Microsoft corporation. Redmond,Washington 98052-6399

[5] htt://www.erlang.se/publications/asn1_to_erlang-paper.ps.

[6] http://knight.cis.temple.edu/~ingargio/cis307/readings/rpc.html.

[7] http://www.cs.wisc.edu/~sschang/OS-Qual/distOS/RPC.htm.

[8]K.Haviland, D.Gray and B.Salama (1999), UNIX System Programming. Addison Wesley Longman, Inc.